General comments

Bloch and Audet introduce PyRaysum, a Python wrapper for the well-established Raysum Fortran code for "Modeling Ray-theoretical Plane Body-wave Propagation in Dipping Anisotropic Media". PyRaysum is an easy-to-use tool that brings several advantages over Raysum, most notably the convenient integration with the scientific Python stack.

My perspective for this review is that of a user who knows fairly little about modeling ray-theoretical body waves in anisotropic media, but is interested in having easy-to-use tools available for exploration of ideas. My review therefore largely focuses on the code itself and its documentation.

Overall, I am impressed with the accessibility and presentation of both the code and the documentation. The examples in the manuscript are convincing and easily reproducible. However, I believe there are some points that should be addressed before being accepted for publication, such as a somewhat confusing section 2, a slightly lacking performance section, and potentially some changes to the code. Below, I give my feedback on the manuscript and code separately.

Comments on the manuscript

39, 245, 247: Suggest to replace the urls by DOIs.

51 - 118: There is some confusion what the modules are and what the user interacts with. From a user viewpoint, `PyRaysum` provides the packages `pyraysum` and `fraysum`. Section 2.1 decribes the package `fraysum`, whereas 2.2 and 2.3 describe the modules of the package `pyraysum`. This leads to some confusion about what is what. In the examples, the user never interacts explictly with `pyraysum.prs` (importing classes is done directly from `pyraysum`) or `pyraysum.plot` (plotting in Listing 2 is done via `.plot()` methods of `Model` and `Seismogram`). I suggest to restructure this section slightly to be explicity that as a user, there are two packages to import from (`pyraysum` and `fraysum`) and then describe them individually. The above is also relevant to the online documentation, where this same confusion arises.

Listing 1: The comment `# ObsPy Stream` suggests that the output `seis1` is an `obspy.Stream`, where only the entries in `seis1.streams` are. See in comments on code below for more details.

124: []; ... looks like a \citep error to me.

182 - 213 + Figure 5: It would be most interesting to see a comparison with Raysum. In the abstract, the authors mention significantly reduced overhead, which implies that their code should be faster than using Raysum in a common pure Fortran workflow. However, during the performance tests the authors do not compare against this. Instead, the authors focus on comparing `fraysum.run_bare()` versus `pyraysum.run()`. I believe this is still useful for the user to inform them about the trade-off between performance and convenience of the two approaches, but a direct Raysum comparison would be far more interesting. Still, the convenience of adapting Raysum to Python is obvious. But this could also help convince users of Raysum to switch to the authors Python implementation in addition to the benefits such as automatic phase labelling.

Listings 4-6: I believe these code snippets related to the performance section are probably not important enough to include in the main text. They could be moved to an appendix or be moved entirely to the online documentation.

224: Ch[r]istoffel equation

Comments on the code

The code was easy to install and runs without any issues. The code examples provided by the authors are straightfoward and instructive. Overall, I was impressed with the presentation and especially the extensive documentation of the code. I have some minor concerns with the naming of classes and that some expected behaviour is missing. Addressing both of these would make the code more "pythonic" and easier to adapt, but is not strictly necessary.

Class naming

The most significant code users will interface with when using PyRaySum are the classes `Model`, `Geometry`, `RC`, and `Seismogram`. The class names `Geometry`, `RC`, and `Seismogram` are a bit confusing.

- It's unfortunate that both `Model` and `Geometry` contain information about the geometry (model=medium geometry, geometry=ray and station geometry). I don't have a great solution for that, but this may be worth thinking about some more.
- Abbreviated class names (here `RC`) are confusing, I suggest to name it RunControl, RunParameters or something similar.
- `Seismogram` contains more information than the seismograms, in fact synthetic seismograms are stored in `.streams`, receiver functions in `.rfs` in addition to all parameters that generated the seismograms/rfs in `.model`, `.geometry`, and `.rc`. Based on the naming, as a regular Python/obpy user, I was expecting `Seimogram` to behave like an `obspy.Stream`. One possible suggestion to tidy this up could be to make `Seismogram` a subclass of `obspy.Stream` with an additional parameter `prs_parameters` that would contain `prs_parameters.model` etc. Synthetic seismograms and receiver functions could then be identified by their channel name (e.g., `SY(Z|N|E)` and `RF(Z|N|E)`, though this would clash with SEED channel naming conventions). A different way could be to rename `Seismogram` to something more descriptive of what this actually is, e.g., `PRSOutput`. This would also help with the confusion the output of `pyraysum.run()` in Listing 1 in the manuscript (See comment above). At the very least, I believe `Seismogram.streams` and `Seismograms.rfs` should probably be `obspy.Stream` subclasses and not lists, and outputs for different rays/stations should be identified by trace name not index in the lists.

Dunder methods

For some classes, the authors have implemented some of the expected dunder methods (e.g., `__str__()` for `Model`), which is great for Python users. I suggest to expand this to other classes and to make it more consistent across classes. Here some suggestions and observed inconsistencies regarding this:

- There are no `__eq__()` methods to check for equality. By default (as is currently), `modl1 == modl2` will check whether they're the same instance of `Model` (i.e., same memory address), not whether the models have the same parameters. This also applies to the other classes.
- `Model` could have a `__add__()` method that would allow to add additional layers to an existing model, as in `model1 += [thickn, rho, vp, vs]`.
- `Model` could have `__getitem__()` and `__setitem__()` methods to allow extracting and changing individual layers instead of the current `change()` method. I'd expect `modl1[0]` to return all parameters of the first layer and `modl1[0] = [thickn, rho, vp, vs]` to change the values of

the first layer and `modl1[0][1] = rho` to change only its density. The down-side of this would be that the user would need to remember that `modl1[0][1]` refers to density. This could be helped by making layers into dictionaries at least for the user-facing logic. I.e, `modl1[0] = [thickn, rho, vp, vs]` would become `modl1[0] = {"thickn":thickn, "rho":rho, "vp":vp, "vs":vs}` and `modl1[0][1] = rho` would become `modl1[0]["rho"] = rho`. This would likely be a fairly significant change to the current logic of the code and requires some more thinking about potential consequences for other parts of the code, but I believe would ultimately benefit the user by being a) more explicit and b) more natural to Python users.

- `Geometry.__str__()` does not have table headers, but `Model.__str__()` does.
- `RC.__str__()` does have a different table layout than `Model`'s and `Geometry`'s strings.
- `Seismogram` does not have any dunder methods, I'd expect at least `__str__()` to represent the most important information about this model run.

Plots

- `Model.plot()`: Unclear what colors in middle plot mean (density? vp? vs? impedance?). Suggest to have same depth axis also on the interfaces plot (right).

REVIEWER B:

This article presents PyRaysum, a Python software for modeling ray-theoretical body-wave propagation in dipping and/or anisotropic layered media based on the Fortran code Raysum by Frederiksen and Bostock, 2000.

PyRaysum was developed to resolve the shortcoming of Raysum, like the formatting of input files, reading and writing the output files, visualizing the output data, and to facilitate its usage for beginners to streamline the modelling approach in optimization or probabilistic search approaches.

I agree with the authors that the modernisation of older codes is an important step for the progress of research. I am little concerned about the possible use that can be done; it is frequent that users are not really aware of what the codes are doing; but this is nothing that the authors can solve.

In my opinion the article is suitable for publication in Seismica; A few points need to be addressed to make it clearer and to prove that it performs as well as or better than the original code:

- Examples:

It is interesting to see that Pyraysum can reproduce the arrival times, (relative) amplitudes and polarities as Raysum, and it's good to see the example against previous work (e.g. Porter et al., 2011). Anyways more significant and simple examples need to be shown in order to convince old users that Pyraysum can really do its job. For example:

Make a model with one dipping interface and another with one inclined interface but with +180 degrees srike and complementary angle. For example Model1: Interface dipping to the E of 30 degrees; and Model 2: Interface dipping to the W of 60 degrees.

And similar for anisotropy: Model1: one horizontal layer with dipping anisotropy, positive anisotropy with symmetry axis striking to the N and inclined of 30. Model 2 one horizontal layer with dipping negative anisotropy having symmetry axis striking to S and inclined of 60.

- Performance:

A very important performance test is to check the time for performing the same inversion with the original Raysum and with Pyraysum. The question is, How much time is the user really saving?

Reply to Comments

> We have now reached a decision regarding your submission to Seismica, "PyRaysum: Software for Modeling Ray-theoretical Plane Body-wave Propagation in Dipping Anisotropic Media". Based on both the reviews received, your manuscript may be suitable for publication after some revisions.
>
> Both the reviewers asked some more discussion about the performance of the PyRaySum package in terms of CPU time, to understand the difference in large-scale application of the code, with respect to the original one. Reviewers also asked for few more tests and examples.

We thank the editor and the reviewers for their time invested in evaluating our manuscript. In response to their suggestions, we have augmented the *Examples* section with more basic usage examples and a new Figure 3. The *Performance* section now contains a paragraph with a comparison against a classic *Raysum* workflow and an updated Figure 6. In response to Reviewer C we also changed the code base where we augmented double-underscore methods and changed naming conventions for a more intuitive user experience. Please find the detailed answers to the reviewers below.

We hope that our revised article is suitable for publication in *Seismica*.

> **Reviewer B**
>
> Bloch and Audet introduce PyRaysum, a Python wrapper for the well-established Raysum Fortran code for "Modeling Ray-theoretical Plane Body-wave Propagation in Dipping Anisotropic Media". PyRaysum is an easy-to-use tool that brings several advantages over Raysum, most notably the convenient integration with the scientific Python stack.
>
> My perspective for this review is that of a user who knows fairly little about modeling ray-theoretical body waves in anisotropic media, but is interested in having easy-to-use tools available for exploration of ideas. My review therefore largely focuses on the code itself and its documentation.
>
> Overall, I am impressed with the accessibility and presentation of both the code and the documentation. The examples in the manuscript are convincing and easily reproducible. However, I believe there are some points that should be addressed before being accepted for publication, such as a somewhat confusing section 2, a slightly lacking performance section, and potentially some changes to the code. Below, I give my feedback on the manuscript and code separately.

We thank the reviewer for their positive evaluation of our work. We have made substantial improvements to the code base that led to an extended and cleaner *User Interface* section. As a response also to Reviewer C we adopted the comparison with *Raysum* and demonstrate the performance gain in an extended *Performance* section and an updated Figure 6.

> **Comments on the manuscript**
>
> 39, 245, 247: Suggest to replace the urls by DOIs.

These URLs refer to the documentation (github pages) that accompanies *PyRaysum* and are not available as DOIs. We prefer to leave them as URLs unless we are directed otherwise by the production editors at Seismica.

> 51 - 118: There is some confusion what the modules are and what the user interacts with. From a user viewpoint, PyRaysum provides the packages pyraysum and fraysum . Section 2.1 decribes the package fraysum , whereas 2.2 and 2.3 describe the modules of the package pyraysum . This leads to some confusion about what is what. In the examples, the user never interacts explictly with pyraysum.prs (importing classes is done directly from pyraysum) or pyraysum.plot (plotting in Listing 2 is done via .plot() methods of Model and Seismogram). I suggest to

restructure this section slightly to be explicity that as a user, there are two packages to import from (pyraysum and fraysum) and then describe them individually. The above is also relevant to the online documentation, where this same confusion arises.

We thank the reviewer for the suggestion. We split up the (now extended) *pyraysum* package into three modules: *frs*, *prs*, and *plot*. *frs* provides functions for the interaction with *fraysum*, *prs* provides the object-oriented interface and *plot* bundles plotting functionalities. Additionally, the essential object-oriented interface can be imported directly from *pyraysum*. We have re-structured Section 2 *User Interface* to resemble this logic more closely, with *fraysum* and *pyraysum* formatted as subsections, *prs*, *frs*, and *plot* as sub-subsections below *pyraysum* and the essential object-oriented interface as paragraphs within the section *pyraysum*. The novice user may now import everything they need to get started from the top package level, while a more experienced user may choose to explore the supplied modules for advanced functions.

Listing 1: The comment # ObsPy Stream suggests that the output seis1 is an obspy.Stream , where only the entries in seis1.streams are. See in comments on code below for more details.

We changed the confusing comment and adapted some of the suggested changes to the code base to make the *PyRaysum* more accessible.

124: []; ... looks like a \citep error to me.

Corrected.

182 - 213 + Figure 5: It would be most interesting to see a comparison with Raysum. In the abstract, the authors mention significantly reduced overhead, which implies that their code should be faster than using Raysum in a common pure Fortran workflow. However, during the performance tests the authors do not compare against this. Instead, the authors focus on comparing fraysum.run_bare() versus pyraysum.run() . I believe this is still useful for the user to inform them about the trade-off between performance and convenience of the two approaches, but a direct Raysum comparison would be far more interesting. Still, the convenience of adapting Raysum to Python is obvious. But this could also help convince users of Raysum to switch to the authors Python implementation in addition to the benefits such as automatic phase labelling.

We compared the execution time of a typical *Rasyum* call with a comparable call to *PyRaysum*, i.e., one that does not include any post-processing or bookkeeping. For small problems, where the disk in-/output overhead is most significant, the speedup is about 11-fold on our machine. For larger problems, where more time is spent on the actual computation, the performance gain reduces to about 2-fold. Note that we have performed the more recent benchmark on a faster server and that we now adopted the *Fortran* compiler flag “-Ofast”, as the original *Rasyum*. These changes again reduced the absolute execution times of our benchmarks. The relative times remained constant.

Listings 4-6: I believe these code snippets related to the performance section are probably not important enough to include in the main text. They could be moved to an appendix or be moved entirely to the online documentation.

We agree and moved them to the appendix.

224: Ch[r]istoffel equation

Corrected.

Comments on the code

The code was easy to install and runs without any issues. The code examples provided by the authors are straightfoward and instructive. Overall, I was impressed with the presentation and especially the extensive

documentation of the code. I have some minor concerns with the naming of classes and that some expected behaviour is missing. Addressing both of these would make the code more "pythonic" and easier to adapt, but is not strictly necessary.

We are happy to hear that our efforts resulted in a good user experience, and we thank the reviewer for insisting on a modern implementation. We have adapted most of the proposed changes and learned some more *Python* on the way.

Class naming

The most significant code users will interface with when using PyRaySum are the classes Model , Geometry , RC , and Seismogram . The class names Geometry , RC , and Seismogram are a bit confusing.

It's unfortunate that both Model and Geometry contain information about the geometry (model=medium geometry, geometry=ray and station geometry). I don't have a great solution for that, but this may be worth thinking about some more.

The class name *Geometry* is inherited from the *Raysum* file with extension *.geom*, which is one of the required input files describing the ray and station geometry. By sticking to this convention, the intent is to make it easier for seasoned *Raysum* users to adapt their existing workflows to our code. In the documentation of the class and throughout the article, we now chose a more precise wording to emphasize that the name refers to the ray and station geometry.

Abbreviated class names (here RC) are confusing, I suggest to name it RunControl, RunParameters or something similar.

We agree and changed the name, for the sake of brevity, to *Control*.

Seismogram contains more information than the seismograms, in fact synthetic seismograms are stored in .streams , receiver functions in .rfs in addition to all parameters that generated the seismograms/rfs in .model , .geometry , and .rc. Based on the naming, as a regular Python/obspy user, I was expecting Seimogram to behave like an obspy.Stream. One possible suggestion to tidy this up could be to make Seismogram a subclass of obspy.Stream with an additional parameter prs_parameters that would contain prs_parameters.model etc. Synthetic seismograms and receiver functions could then be identified by their channel name (e.g., SY(Z|N|E) and RF(Z|N|E) , though this would clash with SEED channel naming conventions). A different way could be to rename Seismogram to something more descriptive of what this actually is, e.g., PRSOutput . This would also help with the confusion the output of pyraysum.run() in Listing 1 in the manuscript (See comment above). At the very least, I believe Seismogram.streams and Seismograms.rfs should probably be obspy.Stream subclasses and not lists, and outputs for different rays/stations should be identified by trace name not index in the lists.

We agree with the reviewer in that the old class name was confusing. We now renamed it *Result*, as to avoid abbreviations. We prefer to have both .streams, as well as .rfs to be lists of obspy.Stream, because list indexing allows the user to associate list elements to the similarly constructed list elements of Geometry. In this way, a seismogram can easily be associated to a specific ray. Using *__getitem__* methods, we now implemented a consistent indexing of *Geometry* and *Result* in terms of ray indices. This concept is introduced in the main text of the article and documented in the code and online tutorials. We adopted the suggested naming of the channels, where we assume a user who is aware of SEED naming conventions is also able to change these as needed.

Dunder methods

For some classes, the authors have implemented some of the expected dunder methods (e.g., __str__() for Model), which is great for Python users. I suggest to expand this to other classes and to make it more consistent across classes. Here some suggestions and observed inconsistencies regarding this

There are no __eq__() methods to check for equality. By default (as is currently), modl1 == modl2 will check whether they're the same instance of Model (i.e., same memory address), not whether the models have the same parameters. This also applies to the other classes.

We thank the reviewer by pointing us towards the power of dunder methods and implemented them widely. There are now *__eq__()* methods for the *Model* and *Geometry* classes where they evaluate to True if all elements in the user attributes that refer to physical quantities are equal.

Model could have a __add__() method that would allow to add additional layers to an existing model, as in model1 += [thickn, rho, vp, vs] .

We implemented the *__add__()* methods for *Model* and *Geometry* and documented their behavior in the documentation and the online tutorials.

Model could have __getitem__() and __setitem__() methods to allow extracting and changing individual layers instead of the current change() method. I'd expect modl1[0] to return all parameters of the first layer and modl1[0] = [thickn, rho, vp, vs] to change the values of the first layer and modl1[0][1] = rho to change only its density. The down-side of this would be that the user would need to remember that modl1[0][1] refers to density. This could be helped by making layers into dictionaries at least for the user-facing logic. I.e, modl1[0] = [thickn, rho, vp, vs] would become modl1[0] = {"thickn":thickn, "rho":rho, "vp":vp, "vs":vs} and modl1[0][1] = rho would become modl1[0]["rho"] = rho . This would likely be a fairly significant change to the current logic of the code and requires some more thinking about potential consequences for other parts of the code, but I believe would ultimately benefit the user by being a) more explicit and b) more natural to Python users.

We implemented *__getitem__()* and *__setitem__()* methods for the *Model* and *Geometry* classes and __getitem__() for the *Result*. We demonstrate the behavior of *Model.__setitem__()* in the expanded *Examples* section. We believe that implementation of these methods substantially improved the intuitiveness of *PyRaysum*.

Geometry.__str__() does not have table headers, but Model.__str__() does.

The *Geometry.__str__()* method now has table headers as well.

RC.__str__() does have a different table layout than Model 's and Geometry 's strings.

The layout was inspired by the format of the *Raysum* parameter file. We now implemented a more human-readable format for onscreen print output and implemented a separate formatting for file output.

Seismogram does not have any dunder methods, I'd expect at least __str__() to represent the most important information about this model run.

The *Results* object now has a *__str__()* representation, as well as a *__len__()* and *__getitem__()*. A *__setitem__()* method did not appear meaningful to us.

Plots

Model.plot(): Unclear what colors in middle plot mean (density? vp? vs? impedance?). Suggest to have same depth axis also on the interfaces plot (right).

The panel now has a colorbar that indicates that the colors relates to *vs*. The same depth extend on the interface plot is hard to achieve, as it requires the interfering *axes("equal")* option, which leads to a poor layout when combined.

Reviewer C

This article presents PyRaysum, a Python software for modeling ray-theoretical body-wave propagation in dipping and/or anisotropic layered media based on the Fortran code Raysum by Frederiksen and Bostock, 2000.

PyRaysum was developed to resolve the shortcoming of Raysum, like the formatting of input files, reading and writing the output files, visualizing the output data, and to facilitate its usage for beginners to streamline the modelling approach in optimization or probabilistic search approaches.

I agree with the authors that the modernisation of older codes is an important step for the progress of research. I am little concerned about the possible use that can be done; it is frequent that users are not really aware of what the codes are doing; but this is nothing that the authors can solve.

In my opinion the article is suitable for publication in Seismica; A few points need to be addressed to make it clearer and to prove that it performs as well as or better than the original code:

We thank for the positive evaluation of our work. We have refined and more thoroughly tested the functions that allow *PyRaysum* to deal with legacy *Raysum* files and hope that it will be useful to new users interested in the receiver function method as well as experienced *Raysum* users.

Examples:

It is interesting to see that Pyraysum can reproduce the arrival times, (relative) amplitudes and polarities as Raysum, and it's good to see the example against previous work (e.g. Porter et al., 2011). Anyways more significant and simple examples need to be shown in order to convince old users that Pyraysum can really do its job. For example:

Make a model with one dipping interface and another with one inclined interface but with +180 degrees srike and complementary angle. For example Model1: Interface dipping to the E of 30 degrees; and Model 2: Interface dipping to the W of 60 degrees.

And similar for anisotropy: Model1: one horizontal layer with dipping anisotropy, positive anisotropy with symmetry axis striking to the N and inclined of 30. Model 2 one horizontal layer with dipping negative anisotropy having symmetry axis striking to S and inclined of 60.

We expanded the *Examples* section, where we now show the suggested instructive examples. We hope that the combination of narration, code examples, and resulting receiver function figures allow to gain some intuition on what *PyRaysum* does. We use the now expanded *Examples* section to showcase the new syntax suggested by Reviewer B to modify subsurface models.

Performance:

A very important performance test is to check the time for performing the same inversion with the original Raysum and with Pyraysum. The question is, How much time is the user really saving?

We compared the run time of a typical *Rasyum* run with a comparable run of *PyRasyum*. The results are described in the second paragraph of the *Performance* section and shown in the updated Figure 6. Our benchmarks indicate that, with our setup, the reduction in computational cost is 11- to 2-fold, depending on model complexity, mainly due to the reading/writing overhead inherent to *Raysum*.

Improvement:

I would like to see the test for anisotropy in the first (upper) layer. Raysum for example cannot handle it, and it would be fantastic if PyRaysum could overcome this issue.

Two examples of Section *3.2 Interactive exploration of receiver functions* feature an anisotropic topmost layer. During the work on *PyRaysum*, the original author of *Raysum*, Andrew Frederiksen, discovered the bug that prevented the topmost layer to be anisotropic. It is now fixed in the most recent version of *Raysum* as well as in *PyRaysum*. To fix a local version of *Raysum*, in raysum.f, search for the line:

```
call anisotroc(aa(1,1,1,1,2),rho(1),p(1,nseg))
```

and exchange the index “2” for “1” in variable “aa”.