Non-reciprocal wavefields (1D) Documentation

Release 28-03-2018

Christian Reinicke, Kees Wapenaar, and Evert Slob

Apr 18, 2018

Contents

1

p-w-domain

1.1 Layered_NRM_p_w module

Routines for modelling wavefields in 1D non-reciprocal media.

Authors Christian Reinicke (c.reinicke@tudelft.nl), Kees Wapenaar (), and Evert Slob ()

`class Layered_NRM_p_w.Layered_NRM_p_w(`*nt*, *dt*, *nr=1*, *dx1=1*, *verbose=False*, *x3vec=array([0.])*, *avec=array([0.])*, *bvec=array([0.])*, *g1vec=array([0.])*, *g3vec=array([0.])*, *p1=None*, *ReciprocalMedium=False*, *AdjointMedium=False*`)`

Bases: *Wavefield_NRM_p_w.Wavefield_NRM_p_w*

is a class to model wavefields in 1.5D (non-)reciprocal media in the ray-parameter frequency domain.

The class Layered_NRM_p_w defines a 1.5D (non-)reciprocal medium and a scalar wavefield. We consider a single horizontal ray-parameter 'p1' and all frequencies that are sampled by the given number of time samples 'nt' and the time sample interval 'dt'.

Parameters **nt** : int

Number of time samples.

dt : int, float

Time sample interval in seconds.

nr : int, optional

Number of space samples.

dx1 : int, float, optional

Space sample interval.

verbose : bool, optional

Set 'verbose=True' to receive feedback in the command line.

x3vec : numpy.ndarray

Vertical spatial vector x_3, for n layers 'x3vec' must have the shape (n,). We define the x_3-axis as downward-pointing. Implicitly, the first value on the x_3-axis is zero (not stored in 'x3vec').

avec : numpy.ndarray

Medium parameter α (real-valued), for n layers 'avec' must have the shape (n,).

bvec : numpy.ndarray

Medium parameter β (real-valued), for n layers 'bvec' must have the shape (n,).

g1vec : numpy.ndarray, optional

Medium parameter γ_1 (real-valued for non-reciprocal media or imaginary-valued for reciprocal media), for n layers 'g1vec' must have the shape (n,).

g3vec : numpy.ndarray, optional

Medium parameter γ_3 (real-valued for non-reciprocal media or imaginary-valued for reciprocal media), for n layers 'g3vec' must have the shape (n,).

p1 : int, float

Horizontal ray-parameter in seconds per metre.

ReciprocalMedium : bool, optional

For non-reciprocal media set 'ReciprocalMedium=False', for reciprocal media set 'ReciprocalMedium=True'.

AdjointMedium : bool, optional

Set 'AdjointMedium=True' to compute scattering coefficients and propagators in an adjoint medium [(a)]. For reciprocal media, the scattering coefficients and propagators are identical in a medium and its adjoint. We have defined the scattering and propagation in the adjoint medium only for flux-normalisation.

Returns class

A class to model a wavefield in a 1.5D non-reciprocal medium in the ray-parameter frequer

- **x3vec**: x_3.
- **avec**: α.
- **bvec**: β.
- **g1vec**: γ_1.
- **g3vec**: γ_3.
- **p1**: Horizontal ray-parameter.
- **ReciprocalMedium**: True for reciprocal media, False for non-reciprocal media.
- **AdjointMedium**: If True, propagation and scatteing are defined in a medium and in its adjoint.
- **p3**: Vertical ray-parameter for positive 'p1'.
- **p3n**: Vertical ray-parameter for negative 'p1'.

Notes

- **We format the data as described below.**
  - Wavefields are saved in an array of dimensions (nf,nr) in the frequency domain and (nt,nr) in the time domain.
  - Wavefields are in the p- ω domain.
  - The zero frequency component is placed at the first index position.
  - If the wavefield is transformed to the time domain, the zero time component is placed at the first index position, followed by nt/2-1 positive time samples and nt/2 negative time samples.
- **For evanescent waves, Kees makes a sign choice for the vertical ray-parameter,**
  - $p_3' = -j\sqrt{p_1^2 - (\alpha\beta + \gamma_1^2 + \gamma_3^2)}$.

  By default, NumPy makes the oppostie sign choice,
  - $p_3' = +j\sqrt{p_1^2 - (\alpha\beta + \gamma_1^2 + \gamma_3^2)}$.

  We stick to the sign choice by NumPy. Thus, we will also change the sign choice for the propagat
  - Kees chose: $\tilde{w}^{\pm} = \exp(-j\omega p_3' \Delta x_3)$.
  - We choose: $\tilde{w}^{\pm} = \exp(+j\omega p_3' \Delta x_3)$.

References

Kees document as soon as it is published.

Examples

```
>>> from Layered_NRM_p_w import Layered_NRM_p_w as LM
>>> import numpy as np
```

```
>>> # Initialise wavefield in a layered non-reciprocal medium
>>> F=LM(nt=1024,dt=0.005,x3vec=np.array([1.1,2.2,3.7]),avec=np.array([1,2,3]),bvec=np.
↪array([0.4,3.14,2]),p1=2e-4,ReciprocalMedium=False)
```

FocusingFunction_p_w(*x3F*, *normalisation='flux'*, *InternalMultiples=True*)

computes the focusing functions between the top surface ($x_3 = 0$) and the focusing depth defined by the input variable 'x3F'. We define the focusing depth just below 'x3F'. Hence, if the focusing depth coincides with an interface the focusing function focuses below that interface.

Parameters **x3F** : int,float

Focusing depth.

normalisation : str, optional

For pressure-normalisation set normalisation='pressure', for flux-normalisation set normalisation='flux'. Until now, this function only models the focusing function for flux-normalisation.

InternalMultiples : bool, optional

To model internal multiples set 'InternalMultiples=True'. To ignore internal multiples set 'InternalMultiples=False'.

Returns dict

Dictionary that contains

- **FP**: Downgoing focusing function.
- **RP**: Reflection response from above.
- **TP**: Transmission response from above.
- **FM**: Upgoing focusing function.
- **RM**: Reflection response from below.
- **TM**: Transmission response from below.
- **FPa**: Downgoing focusing function (adjoint medium).
- **RPa**: Reflection response from above (adjoint medium).
- **TPa**: Transmission response from above (adjoint medium).
- **FMa**: Upgoing focusing function (adjoint medium).
- **RMa**: Reflection response from below (adjoint medium).
- **TMa**: Transmission response from below (adjoint medium).

All medium responses are stored in arrays of shape (nf,1). The variables 'FPa', 'RPa', 'TPa', 'FMa', 'RMa' and 'TMa' are computed only if one sets 'AdjointMedium=True'.

Notes

- **The downgoing focusing funtion $\tilde{F}_1^+$ is computed by inverting the expressions for the transmis**
  $\tilde{F}_{1,n}^+ = \tilde{F}_{1,n-1}^+(\tilde{w}_n^+)^{-1}(1-\tilde{w}_n^+\tilde{R}_{n-1}^\cap\tilde{w}_n^-\tilde{r}_n^\cup)^{-1}(\tilde{t}_n^+)^{-1}$
- **The upgoing focusing function is computed by applying the reflection response $R^\cup$ on the dow**
  $\tilde{F}_{1,n}^- = \tilde{R}^\cup\tilde{F}_{1,n}^+$.

References

Kees document as soon as it is published.

Examples

```
>>> from Layered_NRM_p_w import Layered_NRM_p_w as LM
>>> import numpy as np
```

```
>>> F=LM( nt=1024,dt=0.005,x3vec=np.array([10,150,200]),
>>>       avec=np.array([1,2,3]),bvec=np.array([0.4,3.14,2]),
>>>       g1vec=np.array([0.9,2.1,0.3]),g3vec=np.array([0.7,1.14,0.2]),
>>>       p1=2e-4,ReciprocalMedium=False,AdjointMedium=True )
```

`GreensFunction_p_w`(*x3R*, *x3S*, *normalisation='flux'*, *InternalMultiples=True*)

computes the one-way Green's functions for a receiver and source depth defined by the input variables 'x3R' and 'x3S'. The one-way wavefields are decomposed at the receiver- and at the source-side. We define the receiver and source depths just below 'x3R' and 'x3S', respectively (this is important if the receiver or source depth coincides with an interface).

Parameters **x3R** : int,float

Receiver depth.

x3S : int, float

Source depth.

normalisation : str, optional

For pressure-normalisation set normalisation='pressure', for flux-normalisation set normalisation='flux'.

InternalMultiples : bool, optional

To model internal multiples set 'InternalMultiples=True'. To ignore internal multiples set 'InternalMultiples=False'.

Returns dict

Dictionary that contains

- **GPP**: Green's function $G^{+,+}$ (true medium).
- **GPM**: Green's function $G^{+,-}$ (true medium).
- **GMP**: Green's function $G^{-,+}$ (true medium).
- **GMM**: Green's function $G^{-,-}$ (true medium).
- **GPPa**: Green's function $G^{+,+}$ (adjoint medium).
- **GPMa**: Green's function $G^{+,-}$ (adjoint medium).
- **GMPa**: Green's function $G^{-,+}$ (adjoint medium).
- **GMMa**: Green's function $G^{-,-}$ (adjoint medium).

All medium responses are stored in arrays of shape (nf,1). The variables 'GPPa', 'GPMa', 'GMPa' and 'GMMa' are computed only if one sets 'AdjointMedium=True'.

Notes

- The superscript '+' and '-' refer to downgoing and upgoing waves, respectively.
- The first superscript refers to the wavefield at the receiver-side.
- The second superscript refers to the wavefield at the source-side.

References

Kees document as soon as it is published.

Examples

```
>>> from Layered_NRM_p_w import Layered_NRM_p_w as LM
>>> import numpy as np
```

```
>>> F=LM( nt=1024,dt=0.005,x3vec=np.array([10,150,200]),
>>>       avec=np.array([1,2,3]),bvec=np.array([0.4,3.14,2]),
>>>       g1vec=np.array([0.9,2.1,0.3]),g3vec=np.array([0.7,1.14,0.2]),
>>>       p1=2e-4,ReciprocalMedium=False,AdjointMedium=True )
```

```
>>> G = F.GreensFunction_p_w(x3R=0,x3S=0,normalisation=normalisation,
>>>                          InternalMultiples=InternalMultiples)
>>> RT=F.RT_response_p_w(normalisation=normalisation,
>>>                      InternalMultiples=InternalMultiples)
>>> np.linalg.norm(RT['RP']-G['GMP']
0.0
```

Insert_layer(*x3*, *UpdateSelf=False*)

inserts a transparent interface at the depth level 'x3'. If 'x3' coincides with an interface of the model the model's interface is left unchanged. If 'x3' is a vector it is interpreted as multiple depth levels at which transparent interfaces will be inserted.

Parameters **x3** : int, float, numpy.ndarray

A depth level, or a vector of depth levels, at which a transparent interface will be inserted. The variable 'x3' either must be a scalar, or have the shape (n,). Each element of 'x3' must be real-valued and greater than, or equal to zero.

UpdateSelf : bool, optional

Set 'UpdateSelf=True' to not only output an updated model but also update the 'self' parameters.

Returns dict

Dictionary that contains

- **x3vec**: Updated depth vector.
- **avec**: Updated α vector.
- **bvec**: Updated β vector.
- **g1vec**: Updated γ_1 vector.
- **g3vec**: Updated γ_3 vector.
- **p3**: Updated $p_3(p_1)$ vector.
- **p3n**: Updated $p_3(-p_1)$ vector.

All medium parameter vectors are stored in arrays of shape (n,).

Examples

```
>>> from Layered_NRM_p_w import Layered_NRM_p_w as LM
>>> import numpy as np
```

```
>>> # Initialise a wavefield in a 1D reciprocal medium
>>> F=LM(nt=1024,dt=0.005,x3vec=np.array([10,150,200]),avec=np.array([1,2,3]),
>>>      bvec=np.array([0.4,3.14,2]),g1vec=np.array([0.9,2.1,0.3]),
>>>      g3vec=np.array([0.7,1.14,0.2]),p1=2e-4)
```

```
>>> # Insert a transparent layer at x3=1
>>> out=F.Insert_layer(x3=1,UpdateSelf=False)
```

```
>>> # Updated depth vector
>>> out['x3vec']
array([  1,  10, 150, 200])
```

```
>>> # Updated alpha vector
>>> out['avec']
array([1, 1, 2, 3])
```

L_eigenvectors_p_w(*beta=None*, *g3=None*, *p3=None*, *p3n=None*, *normalisation='flux'*)

computes the eigenvector matrix 'L' and its inverse 'Linv', either in flux- or in pressure-normalisation for a single vertical ray-parameter 'p3' inside a homogeneous layer. If 'AdjointMedium=True', **L_eigenvectors_p_w** also computes the eigenvector matrix in the adjoint medium 'La' and its inverse 'Lainv'.

Parameters **beta** : int, float

Medium parameter β (real-valued).

g3 : int, float

Medium parameter γ_3.

p3 : int, float

Vertical ray-parameter p_3 for a positive horizontal ray-parameter p_1.

p3n : int, float, optional (required if 'AdjointMedium=True')

Vertical ray-parameter p_3 for a negative horizontal ray-parameter p_1.

normalisation : str, optional

For pressure-normalisation set normalisation='pressure', for flux-normalisation set normalisation='flux'.

Returns dict

Dictionary that contains

- **L**: The eigenvector matrix.
- **Linv**: The inverse of the eigenvector matrix.
- **La**: The eigenvector matrix (adjoint medium).
- **Lainv**: The inverse of the eigenvector matrix (adjoint medium).

All eigenvector matrices are stored in a in a (2x2)-array.

Notes

- The eigenvector matrix 'L' and its inverse 'Linv' are different for reciprocal and non-reciprocal media.

- For reciprocal media, the eigenvectors of the adjoint medium are identical to the eigenvectors of the true medium.
- We have defined the eigenvectors of the adjoint medium only for flux-normalisation.

References

Kees document as soon as it is published.

Examples

```
>>> from Layered_NRM_p_w import Layered_NRM_p_w as LM
>>> import numpy as np
```

```
>>> # Initialise wavefield in a layered non-reciprocal medium
>>> F=LM( nt=1024 , dt=0.005 , x3vec=np.array([1.1,2.2,3.7]) , avec=np.array([1,2,3]) ,␣
↪bvec=np.array([0.4,3.14,2]) , g1vec=np.array([0.9,2.1,0.3]) , g3vec=np.array([0.7,1.
↪14,0.2]) , p1=2e-4 , ReciprocalMedium=False )
```

```
>>> # Compute eigenvectors in flux-normalisation
>>> Lvecs=F.L_eigenvectors_p_w(beta=0.1,g3=0.4,p3=2e-4,normalisation='flux')
```

```
>>> # Eigenvector matrix
>>> L = Lvecs['L']
([[15.8113883 +0.j, 15.8113883 +0.j],[ 0.03162278+0.j, -0.03162278+0.j]])
```

```
>>> # Inverse eigenvector matrix
>>> Linv = Lvecs['Linv']
([[  0.03162278+0.j,  15.8113883 +0.j],[  0.03162278+0.j, -15.8113883 -0.j]])
```

RT_p_w(*beta_u=None*, *g3_u=None*, *p3_u=None*, *p3n_u=None*, *beta_l=None*, *g3_l=None*, *p3_l=None*, *p3n_l=None*, *normalisation='flux'*)

computes the scattering coefficients at an horizontal interface, either in flux- or in pressure-normalisation. The variables with subscript 'u' refer to the medium parameters in the upper half-space, the variables with subscript 'l' refer to the medium parameters in the lower half-space. We consider a single horizontal ray-parameter p_1, which is associated with a vertical ray-parameter 'p3_u' in the upper half-space and 'p3_l' in the lower half-space. If one sets 'AdjointMedium=True', **RT_p_w** also computes the scattering coefficients in the adjoint medium.

Parameters **beta_u** : int, float

Medium parameter β (real-valued) (upper half-space).

g3_u : int, float

Medium parameter γ_3 (upper half-space).

p3_u : int, float

Vertical ray-parameter p_3 for a positive horizontal ray-parameter p_1 (upper half-space).

p3n_u : int, float, optional (required if 'AdjointMedium=True')

Vertical ray-parameter p_3 for a negative horizontal ray-parameter p_1 (upper half-space).

beta_l : int, float

Medium parameter β (real-valued) (lower half-space).

g3_l : int, float

Medium parameter γ_3 (lower half-space).

p3_l : int, float

Vertical ray-parameter p_3 for a positive horizontal ray-parameter p_1 (lower half-space).

p3n_l : int, float, optional (required if 'AdjointMedium=True')

Vertical ray-parameter p_3 for a negative horizontal ray-parameter p_1 (lower half-space).

normalisation : str, optional

For pressure-normalisation set normalisation='pressure', for flux-normalisation set normalisation='flux'.

Returns dict

Dictionary that contains

- **rP**: Reflection coefficient from above.
- **tP**: Transmission coefficient from above 'tP'.
- **rM**: Reflection coefficient from below.
- **tM**: Transmission coefficient from below.
- **rPa**: Reflection coefficient from above (adjoint medium).
- **tPa**: Transmission coefficient from above (adjoint medium).
- **rMa**: Reflection coefficient from below (adjoint medium).
- **tMa**: Transmission coefficient from below (adjoint medium).

All scattering coefficients are stored as scalars.

Notes

- For reciprocal media, the scattering coefficients of the adjoint medium are identical to the scattering coefficients of the true medium.
- We have defined the scattering coefficients of the adjoint medium only for flux-normalisation.

References

Kees document as soon as it is published.

Examples

```
>>> from Layered_NRM_p_w import Layered_NRM_p_w as LM
>>> import numpy as np
```

```
>>> # Create wavefield F for positive horizontal ray-parameter p1
>>> F=LM(nt=1024,dt=0.005,x3vec=np.array([1.1,2.2,3.7]),avec=np.array([1,2,3]),bvec=np.
↪array([0.4,3.14,2]),g1vec=np.array([0.9,2.1,0.3]),g3vec=np.array([0.7,1.14,0.2]),
↪p1=2e-4,ReciprocalMedium=False,AdjointMedium=True)
>>> ScatCoeffs = F.RT_p_w(beta_u=F.bvec[0],g3_u=F.g3vec[0],p3_u=F.p3[0],p3n_u=F.p3n[0],
↪beta_l=F.bvec[1],g3_l=F.g3vec[1],p3_l=F.p3[1],p3n_l=F.p3n[1],normalisation='flux')
>>> rplus = ScatCoeffs['rP']
(0.8620013269525346+0.5069060192304582j)
```

```
>>> # Create wavefield Fn for negative horizontal ray-parameter p1
>>> Fn=LM(nt=1024,dt=0.005,x3vec=np.array([1.1,2.2,3.7]),avec=np.array([1,2,3]),bvec=np.
↪array([0.4,3.14,2]),g1vec=np.array([0.9,2.1,0.3]),g3vec=np.array([0.7,1.14,0.2]),p1=-
↪2e-4,ReciprocalMedium=False,AdjointMedium=True)
>>> ScatCoeffsn = Fn.RT_p_w(beta_u=Fn.bvec[0],g3_u=Fn.g3vec[0],p3_u=Fn.p3[0],p3n_u=Fn.
↪p3n[0],beta_l=Fn.bvec[1],g3_l=Fn.g3vec[1],p3_l=Fn.p3[1],p3n_l=Fn.p3n[1],normalisation=
↪'flux')
```

```
>>> # In non-reciprocal media, for flux-normalisation, the reflection coefficients
>>> # in true medium for positive horizontal ray-parameter p1
>>> # and in adjoint medium for negative horizontal ray-parameter p1
>>> # are identical:
>>> np.abs(ScatCoeffs['rP']-ScatCoeffsn['rPa'])
0.0
```

RT_response_p_w(*x3vec=None*, *avec=None*, *bvec=None*, *g1vec=None*, *g3vec=None*, *normalisation='flux'*, *InternalMultiples=True*)

computes the reflection and transmission responses from above and from below. If medium parameters are given the computed responses are associated with the given parameters medium. Otherwise, the medium parameters defined in **Layered_NRM_p_w** are used.

The medium responses are associated to measurements at $x_3 = 0$ and at x_3 = 'x3vec[-2]' $+\epsilon$, where ϵ is an infinitesimally small positive constant. Hence, the propagation from $x_3 = 0$ to the shallowest interface is included. However, the propagation through the deepest layer is excluded.

Parameters **x3vec** : numpy.ndarray, optional

Vertical spatial vector x_3, for n layers 'x3vec' must have the shape (n,). We define the x_3-axis as downward-pointing. Implicitly, the first value on the x_3-axis is zero (not stored in 'x3vec').

avec : numpy.ndarray, optional

Medium parameter α (real-valued), for n layers 'avec' must have the shape (n,).

bvec : numpy.ndarray, optional

Medium parameter β (real-valued), for n layers 'bvec' must have the shape (n,).

g1vec : numpy.ndarray, optional

Medium parameter γ_1 (real-valued for non-reciprocal media or imaginary-valued for reciprocal media), for n layers 'g1vec' must have the shape (n,).

g3vec : numpy.ndarray, optional

Medium parameter γ_3 (real-valued for non-reciprocal media or imaginary-valued for reciprocal media), for n layers 'g3vec' must have the shape (n,).

normalisation : str, optional

For pressure-normalisation set normalisation='pressure', for flux-normalisation set normalisation='flux'.

InternalMultiples : bool, optional

To model internal multiples set 'InternalMultiples=True'. To ignore internal multiples set 'InternalMultiples=False'.

Returns dict

Dictionary that contains

- **RP**: Reflection response from above.
- **TP**: Transmission response from above.
- **RM**: Reflection response from below.
- **TM**: Transmission response from below.
- **RPa**: Reflection response from above (adjoint medium).
- **TPa**: Transmission response from above (adjoint medium).
- **RMa**: Reflection response from below (adjoint medium).
- **TMa**: Transmission response from below (adjoint medium).

All medium responses are stored in arrays of shape (nf,1). The variables 'RPa', 'TPa', 'RMa' and 'TMa' are computed only if one sets 'AdjointMedium=True'.

References

Kees document as soon as it is published.

Examples

```
>>> from Layered_NRM_p_w import Layered_NRM_p_w as LM
>>> import numpy as np
```

```
>>> # Initialise a wavefield in a 1D reciprocal medium
>>> # Here, the parameters are chosen such that the wavefield is purely propagating␣
↪(not evanescent)
>>> F=LM(nt=1024,dt=0.005,x3vec=np.array([100,500,1000,1010]),avec=np.array([5,2,3,4]),
↪bvec=np.array([0.4,3.14,2,1.5]),g1vec=np.array([0.9,2.1,0.3,0.25]),g3vec=np.array([0.
↪7,1.14,0.2,0.3]),p1=2e-4,ReciprocalMedium=False)
```

```
>>> # Model the medium responses
>>> Responses=F.RT_response_p_w(normalisation='flux',InternalMultiples=True)
>>> # Here are your first medium responses:
>>> Rplus = Responses['RP']
>>> Tplus = Responses['TP']
```

```
>>> # Verify if conservation of energy is satisfied
>>> Rplus.conj()*Rplus+Tplus.conj()*Tplus
([[1.+0.j], [1.+0.j], ..., [1.+0.j]])
```

`W_propagators_p_w`(*p3=None*, *p3n=None*, *g3=None*, *dx3=None*, *w=None*)

computes the downgoing propagator 'wP' and the upgoing progagator 'wM' for a single vertical ray-parameter 'p3' and a vertical distance 'dx3' (downward pointing x_3-axis).

Parameters **p3** : int, float

Vertical ray-parameter p_3 for a positive horizontal ray-parameter p_1.

p3n : int, float, optional (required if 'AdjointMedium=True')

Vertical ray-parameter p_3 for a negative horizontal ray-parameter p_1.

g3 : int, float

Medium parameter γ_3.

dx3 : int, float

Vertical propagation distance Δx_3 (downward pointing x_3-axis).

w : int, float

Frequency ω in radians. By default the propagators are computed for all sampled (positive) frequencies.

Returns dict

Dictionary that contains

- **wP**: Downward propagator $\tilde{w}^{+}$.
- **wM**: Upward propagator $\tilde{w}^{-}$.
- **wPa**: Downward propagator $\tilde{w}^{+(a)}$ (adjoint medium).
- **wMa**: Upward propagator $\tilde{w}^{-(a)}$ (adjoint medium).

All propagators are stored either in an arrays of shape (nf,1), or as a scalar (if the variable'w' is set). The variables 'wPa' and 'wMa' are computed only if one sets 'AdjointMedium=True'.

References

Kees document as soon as it is published.

Examples

```
>>> from Layered_NRM_p_w import Layered_NRM_p_w as LM
>>> import numpy as np
```

```
>>> F=LM(nt=1024,dt=0.005,x3vec=np.array([1.1,2.2,3.7]),avec=np.array([1,2,3]),bvec=np.
↪array([0.4,3.14,2]),g1vec=np.array([0.9,2.1,0.3]),g3vec=np.array([0.7,1.14,0.2]),
↪p1=2e-4,ReciprocalMedium=False)
>>> W=F.W_propagators_p_w( p3=F.p3[0] , p3n=F.p3n[0] , g3=F.g3vec[0] , dx3=F.x3vec[1]-F.
↪x3vec[0])
```

```
>>> W['wP']
[[ 1.00000000e+00+0.        j],[0.24690276+0.34159244j],...,[1.07001473e-192-1.
↪48037608e-192j]]
```

```
>>> # The propagator has nf samples because it is computed only for positive frequencies
>>> W['wP'].shape
(513, 1)
```

```
>>> # AdjointMedium=False, hence, we expect output 'None'
>>> W['wPa']
```

1.2 Wavefield_NRM_p_w module

Routines for modelling wavefields in 1D non-reciprocal media.

Authors Christian Reinicke (c.reinicke@tudelft.nl), Kees Wapenaar (), and Evert Slob ()

Copyright Christian Reinicke (c.reinicke@tudelft.nl), Kees Wapenaar (), and Evert Slob ()

class Wavefield_NRM_p_w.**Wavefield_NRM_p_w**(*nt*, *dt*, *nr=1*, *dx1=1*, *verbose=False*)

Bases: object[1]

is a class to define a scalar wavefield in the ray-parameter frequency domain.

The class Wavefield_NRM_p_w defines the parameters of a scalar wavefield in a 1.5D (non-)reciprocal medium. We consider a single ray-parameter 'p1' and all frequencies that are sampled by the given number of time samples 'nt' and the time sample interval 'dt'.

Parameters **nt** : int

Number of time samples.

dt : int, float

Time sample interval in seconds.

nr : int, optional

Number of space samples.

dx1 : int, float, optional

Space sample interval.

verbose : bool, optional

Set 'verbose=True' to receive feedback in the command line.

Returns class

A class to define a wavefield in a 1.5D non-reciprocal medium in the ray-parameter frequ

- **nt**: Number of time samples.
- **dt**: Time sample interval in seconds.
- **nr**: Number of space samples.
- **dx1**: Number of space samples.
- **verbose**: If one sets 'verbose=True' feedback will be output in the command line.
- **nf**: Number of positive time samples $= 0.5nt + 1$.

[1] https://docs.python.org/3/library/functions.html#object

- **nk**: Number of positive space samples $= 0.5nr + 1$.

Notes

We format the data as described below.

- Wavefields are saved in an array of dimensions (nf,nr) in the frequency domain and (nt,nr) in the time domain.
- Wavefields are in the p- ω domain.
- The zero frequency component is placed at the first index position.
- If the wavefield is transformed to the time domain, the zero time component is placed at the first index position, followed by nt/2-1 positive time samples and nt/2 negative time samples.

Examples

```
>>> # Initialise a wavefield class
>>> from Wavefield_NRM_p_w import Wavefield_NRM_p_w as WF
>>> F=WF(nt=1024,dt=0.005)
```

`Dw()`

returns frequency sampling interval $\Delta\ \omega$ in radians.

The frequency sampling interval is defined by the time sampling interval Δt and the number of time samples 'nt'.

Returns float

Frequency sample interval in radians $\Delta\omega = \frac{2\pi}{\Delta t\ nt}$

Examples

```
>>> from Wavefield_NRM_p_w import Wavefield_NRM_p_w as WF
>>> F=WF(nt=1024,dt=0.005)
>>> F.Dw()
1.227184630308513
```

`PT2PW(`*array_pt*, *NumPy_fft_Sign_Convention=False*`)`

applies an inverse Fourier transform from the p_1-t domain to the p_1-ω domain.

We assume that the time domain signal is real-valued (p_1-t domain). Therefore, we use the NumPy function `numpy.fft.rfft`.

Parameters **array_pt** : numpy.ndarray

Real-valued array in the p_1-t domain, shape (nt,nr).

NumPy_fft_Sign_Convention : bool, optional

Set 'NumPy_fft_Sign_Convention=True' if Numpy's sign convention is used for the Fourier transform (negative sign in the exponential of the forward Fourier transform).

Returns numpy.ndarray

Array in the p_1-ω domain, shape (nf,nr).

Notes

In the sub-class **Layered_NRM_p_w** we define the wavefield propagators with a positive sign, $\tilde{w}^{\pm} = \exp(+j\omega p_3' \Delta x_3)$. Thus, we implicitly assume that the forward Fourier transfrom is defined with a positive sign in the exponential function, which is why we set by default 'NumPy_fft_Sign_Convention=False'.

`PW2PT(`*array_pw*, *NumPy_fft_Sign_Convention=False*`)`
applies an inverse Fourier transform from the p_1-ω domain to the p_1-t domain.

We assume that the time domain signal is real-valued (p_1-t domain). Therefore, we use the NumPy function `numpy.fft.irfft`.

Parameters **array_pw** : numpy.ndarray

Array in the p_1-ω domain, shape (nf,nr).

NumPy_fft_Sign_Convention : bool, optional

Set 'NumPy_fft_Sign_Convention=True' if Numpy's sign convention is used for the inverse Fourier transform (positive sign in the exponential of the inverse Fourier transform).

Returns numpy.ndarray

Real-valued array in the p_1-t domain, shape (nt,nr).

Notes

In the sub-class **Layered_NRM_p_w** we define the wavefield propagators with a positive sign, $\tilde{w}^{\pm} = \exp(+j\omega p_3' \Delta x_3)$. Thus, we implicitly assume that the inverse Fourier transfrom is defined with a negative sign in the exponential function, which is why we set by default 'NumPy_fft_Sign_Convention=False'.

`T_X_grid()`
returns two time-space meshgrids.

Returns dict

Dictionary that contains a meshgrid

- **Tgrid**: with the time vector *tvec* (the zero time sample is placed in the center) along the 1st dimension, and nr copies of it along the 2nd dimension.
- **Xgrid**: with the offset vector *xvec* (the zero offset sample is placed in the center) along the 2nd dimension, and nt copies of it along the 1st dimension.
- **Tgridfft**: with the time vector *tvecfft* (the zero time sample is placed at the first index position) along the 1st dimension, and nr copies of it along the 2nd dimension.
- **Xgridfft**: with the offset vector *xvecfft* (the zero offset sample is placed at the first index position) along the 2nd dimension, and nt copies of it along the 1st dimension.

All output arrays have the shape (nt,nr).

Examples

```
>>> from Wavefield_NRM_p_w import Wavefield_NRM_p_w as WF
>>> F=WF(nt=1024,dt=0.005,nr=512,dx1=10)
>>> F.T_X_grid()['Tgridfft'][:,0]
array([ 0.   ,  0.005,  0.01 , ..., -0.015, -0.01 , -0.005])
```

`Tvec()`

returns a vector of all time samples t in seconds.

Returns dict

Dictionary that contains the time vector,

- **tvec**: zero time placed at the center.
- **tvecfft**: zero time placed at the first index position.

Both vectors have the shape (nt,1).

Examples

```
>>> from Wavefield_NRM_p_w import Wavefield_NRM_p_w as WF
>>> F=WF(nt=1024,dt=0.005,nr=512,dx1=10)
>>> F.Tvec()['tvec']
array([[-2.56 ], ..., [ 2.555]])
```

`W_X_grid()`

returns two frequency-space meshgrids.

Returns dict

Dictionary that contains a meshgrid

- **Wgrid**: with the frequency vector *wvec* (the zero frequency sample is placed in the center) along the 1st dimension, and nr copies of it along the 2nd dimension.
- **Xgrid**: with the offset vector *xvec* (the zero offset sample is placed in the center) along the 2nd dimension, and nt copies of it along the 1st dimension.
- **Xgridfft**: with the offset vector *xvecfft* (the zero offset sample is placed at the first index position) along the 2nd dimension, and nt copies of it along the 1st dimension.

All output arrays have the shape (nf,nr).

Examples

```
>>> from Wavefield_NRM_p_w import Wavefield_NRM_p_w as WF
>>> F=WF(nt=1024,dt=0.005,nr=512,dx1=10)
>>> F.W_X_grid()['Wgrid'][:,0]
array([ 0.        ,  1.22718463,  2.45436926, ..., 628.3185307179587])
```

`Wvec()`

returns a vector of all frequency samples ω in radians.

Returns numpy.ndarray

Frequency vector, zero frequency is placed at the first index position. The vector has the shape (nf,1).

Examples

```
>>> from Wavefield_NRM_p_w import Wavefield_NRM_p_w as WF
>>> F=WF(nt=1024,dt=0.005)
>>> F.Wvec()
array([[0.], ..., [628.31853072]])
```

`Xvec()`

returns a vector of all spatial samples x in metres.

Returns dict

Dictionary that contains the offset vector,

- **xvec**: zero offset placed at the center.
- **xvecfft**: zero offset placed at the first index position.

Both vectors have the shape (nt,1).

Examples

```
>>> from Wavefield_NRM_p_w import Wavefield_NRM_p_w as WF
>>> F=WF(nt=1024,dt=0.005,nr=512,dx1=10)
>>> F.Xvec()['xvec']
array([[-2560.], ...,[ 2550.]])
```

2

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index