Stanford Exploration Project, Report 97, July 8, 1998, pages 1–??

Genkir3D: A toolkit for Kirchhoff imaging

Biondo L. Biondi[1]

ABSTRACT

Genkir3D is a software package that facilitates the implementation of new 3-D Kirchhoff algorithms. It enables the user to specify a new Kirchhoff operator by only writing a function that computes the kinematics and the amplitudes of the summation surfaces, in addition to the maximum data frequency for anti-aliasing. All the practical issues related to data interpolation, data filtering, geometry handling, and limited memory usage, are efficiently and flexibly taken care by the package. The input and the output are data sets in Sep3D data format, and therefore can have arbitrary trace geometry. The package is developed in Fortran 90.

INTRODUCTION

At the basis of a vast class of 3-D imaging algorithms there is a summation of seismic data over multi-dimensional surfaces. While this summation can be usually expressed mathematically in compact notations, its implementation on real seismic data can be complex and time consuming because of issues related with irregular sampling of the data and the limitations in computer memory that make it impossible to process a whole real data set in core.

The goal of the **Genkir3D** package is to help the development and testing of new methods to image 3-D seismic data by allowing the researcher to focus on the specifics of new operators, by hiding the mundane issues related with data geometries, anti-aliasing filtering, and memory allocation. The package automatically reads and write the data in the Sep3D data format that efficiently handle irregularly and regularly sampled seismic data.

The package is called by calling a Fortran 90 subroutine with the appropriate parameters. An example of this function call is:

```
call genkir3d(param=gen_param,  &
  out_geom=out_geom, &
  out_geom_block=out_geom_block, &
  trace_interp_real=trace_interp_lin_real, &
  spat_interp_real=spat_interp_lin_near_real, &
  map_amp=dmo3d_map_amp,  &
```

[1]**email:** biondo@sep.stanford.edu

```
oper_domain=dmo3d_domain)
```

The user controls the particular operator to be applied to the data by specifying the functions that are internally called by the package to compute the summation surface. The name of these functions are the values of the `map_amp` and `oper_domain` arguments. In our example `map_amp=dmo3d_map_amp` and `oper_domain=dmo3d_domain`, and thus a 3-D DMO will be applied to the data. The implementation of the desired integral operator is fine-tuned by setting the values of the fields in the data structure that is passed to the Genkir3D package as the `param` argument. Since both the data and the output space are sampled, temporal and spatial interpolation are important components of a 3-D integral operator. The type of interpolator to be used can be specified by passing the name of the desired interpolators with the `trace_interp_real` and `spat_interp_real` parameters. Standard nearest neighborhood and linear interpolators are provided by the package, but the user can use his/her own new interpolators, by providing an appropriate function. The geometry of the output space, and its blocking in memory, are specified by `out_gem` and `out_gem_block`.

BASIC CONCEPTS

The Genkir3D package simplifies the implementation of several integral operators because it implements the features common to a wide class of integral operators, but, at the same time, it enables the user to specify the distinguishing features of the desired operator. Mathematically, integral operators are mainly differentiated by the shape of the summation surfaces and the values of the amplitude terms associated with the summation surface. Because both the input and output spaces are sampled, antialiasing is another important component of integral operators. Users of Genkir3D can specify all these characteristics of integral operators by writing appropriate functions. Another practical, but important, differentiation, is the "optimal" type of spatial mapping between the input and the output data. Some operators are best applied as spraying operators, while others are best applied as gathering operators.

This section explains the basic concepts on how to specify the summation surfaces and the spatial mapping. We start with the description of the available spatial mappings because the specifications of the summation surfaces depend on the chosen spatial mapping.

SPATIAL MAPPING BETWEEN INPUT AND OUTPUT

The desired spatial mapping is specified by the value of the field `spat_map` in the parameter data structure The following three kind of mapping are supported by the package:

1. Trace-to-trace mapping: `spat_map=trace_to_trace`
2. Output-oriented (Gathering): `spat_map=gather`
3. Input-oriented (Spraying): `spat_map=spray`

Because the optimal data-flow of both the input and the output are closely related to the spatial mapping, the mapping also determines the data flow to be used.

Trace-to-trace mapping

This is the simplest kind of mapping between input and output. It means that each input trace maps into a single output trace with the same geometry. A well known example is the NMO transformation.

In this case, neither of the output geometry data structures (`out_geom`) and (`out_geom_block`) need to be passed to `genkir3d` because in this case the output geometry is "congruent" with the input geometry and thus does not need to be specified. The data flow is also very simple; both the input and the output data are accessed sequentially and buffered in memory. The size of the buffers are automatically determined by the software as a function of the memory limits specified by the user.

Gathering mapping

This kind of mapping is ideal when the operator is function of the output location through a complex relationship that is usually precomputed and appropriately stored in tables. Important example of such operators are Kirchhoff depth migrations, that need tables of precomputed Green functions. The associated data flow is optimal for accessing the mapping tables in small subsets, and maximizing their reuse and consequently minimizing the associated I/Os.

This output-oriented mapping requires that the data structure that describes the geometry of the whole output (`out_geom`) is passed to `genkir3d`. The access to the output data can be blocked along all the output space dimensions; that is the midpoint axes and the offset and azimuth axes. The geometry of the blocks is specified with the argument `out_geom_block`. To localize the access to the mapping tables, the inner loops over the output space are in the midpoint coordinates (`cmp_x` and `cmp_y`) of `out_geom_block`.

A Sep3d data set can be either gridded or ungridded. An ungridded data set is accessed sequentially. The data are read multiple times, as many times as the number of output blocks. A gridded data set is accessed selectively according to the offset and the azimuth of the output block being currently built. Furthermore, for efficiency purposes, the maximum aperture of the operator along the midpoint dimensions can be specified in the param data structure, limiting the ranges of midpoints of the input data being accessed for each output block.

Spraying mapping

This kind of mapping is ideal when the operator is defined analytically, or quasi-analytically, as a function of few parameters of the input and output geometries. Examples of this kind of operators are DMO, AMO, and stacking. Often the output space of these operators is more effectively defined in an input-centered coordinate system. Therefore, when the spraying mapping is applied, the data are first mapped into an intermediate output space, and then spatially interpolated and summed into the final output space. Therefore, two functions must be defined: the oper_domain function that defines the operator domain, and the map_amp that describes the integration surface.

This mapping requires that the data structure that describes the geometry of the whole output space (out_geom) is passed to genkir3d. The access to the output data can be blocked along the offset and azimuth dimensions by specifying e geometry of the blocks with the argument out_geom_block. To optimize the access to the input data, the inner loops over the output space are in the offset and azimuth coordinates (aoffset and azimuth) of out_geom_block.

An ungridded data set is accessed sequentially. The data are read multiple times, as many times as the number of output blocks. A gridded data set is accessed selectively according to the offset and azimuth of the output block being currently built. Each data trace is read only once.

OPERATOR DEFINITION

Ideally the user of the package would need only to define the characteristics of the particular operators he intends to test. The kind of characteristics depend on the complexity of the operator and on the type of spatial mapping that is optimal.

In the simple trace-to-trace mapping the operator is simply described by a transformation of the time axis. In the more general case, an integral operator is defined by a transformation of the time axis for each pair of input and output traces. The generic interface for the mapping function map_amp is:

```
function map_amp (want_info,inv_map,inv_amp,inp_geom,out_geom, &
      first_samp, last_samp, map, amp, inp_cent_coord) result (info)
type(genkir3d_info_type) :: info
logical, intent(in)      :: want_info,inv_map,inv_amp
type (seis3d_geom_type), intent(in) :: inp_geom
type (seis3d_geom_type), intent(in) :: out_geom
integer, dimension(:,:), intent(out) :: first_samp,last_samp
real,  dimension (:,:,:), intent(out)  :: map,amp,max_freq
```

```
real, pointer, dimension (:,:,:)  :: inp_cent_coord
```

For each pair of input and output traces, `map_amp` must compute the mapping function of the input time axis into the output time axis, (`map`) and by its corresponding amplitude term (`amp`). The lower and upper limit of the mapping are respectively described by the `first_samp` and `last_samp` arrays. To avoid aliasing artifacts the input traces can be filtered with a low-pass filter before being summed into the output space. The high-frequency cutoff of this low-pass filtering must also be computed by `map_amp` and stored in the `max_freq` array.

As discussed above, when the spraying spatial mapping is applied, the data are first sprayed into an intermediary output space. The array `inp_cent_coord` contains the coordinates of this intermediate output space. It is used by the `map_amp` function and it is initialized by the operator-domain definition function `oper_domain`. The generic interface for the operator-domain function is:

```
function oper_domain (inv_map,inv_amp,inp_geom,out_geom, &
                      inp_cent_coord, out_coord) result (stat)

integer                 :: stat
logical, intent(in)     :: inv_map,inv_amp
type (seis3d_geom_type), intent(in) :: inp_geom
type (seis3d_geom_type), intent(inout) :: out_geom
real, pointer,  dimension (:,:,:)  :: inp_cent_coord
real, pointer,  dimension (:,:,:)  :: out_coord
```

The array `inp_cent_coord` must be filled with the coordinates of the spraying domain, in input-centered coordinate system. The coordinates of the spraying domain in the output-space coordinate system is stored in the `out_geom` data structure.

DATA STRUCTURES

Genkir3D uses data structures (*derived types* in Fortran 90) to store information on:

1. Parameters
2. Seismic data
3. Operator status and memory requirements

DERIVED TYPES FOR PARAMETERS

Derived type: genkir3d_param_type

This section describes the data-structure for the parameters needed to run Genkir3D. The user does not need to initialize all the parameters. The data structure is automatically initialized to reasonable defaults when the function genkir3d_init is called in the main program.

```
type genkir3d_param_type
  logical :: inv_map,inv_amp,adj,add
  logical :: verb,debug1,debug2,pipe_inp,pipe_out
  logical :: seq_inp_acc,seq_out_acc,out_in_lines
  character(len=128) :: tag_inp,tag_out
  character(len=7) :: inp_type,out_type
  integer :: n_threads,max_memory,non_data_memory
  integer :: n_t_bpass,n_freq_bpass,oversamp_bpass
  integer :: n_t
  real :: t_0,d_t
  real :: min_freq_bpass,max_freq_bpass
end type  genkir3d_param_type
```

Scalar variables		
Name	*Description*	*Default*
inv_map	Apply mapping of inverse operator	.false.
inv_amp	Apply amplitudes of inverse operator	.false.
adj	Apply adjoint operator	.false.
add	Add to output adjoint operator	.false.
verb	Verbose mode	.false.
debug1	First level of debugging (more verbose)	.false.
debug2	Second level of debugging (even more verbose)	.false.
out_in_lines	The (regular) output is written in line order	.false.
tag_inp	Tag of input file	in
tag_out	Tag of input file	out
inp_type	Type of input data (any, float, complex)	any
out_type	Type of output data (same, float, complex)	same
spat_map	Spatial mapping (trace_to_trace, spray, gather)	spray
max_apert_cmpx	Maximum operator aperture along cmpx	huge(1.)
max_apert_cmpy	Maximum operator aperture along cmpy	huge(1.)
n_t	Length of output time axis (when output is irregular)	-huge(1)
t_0	Origin of output time axis (when output is irregular)	-huge(1.)
d_t	Sampling rate of output time axis (when output is irregular)	-huge(1.)
n_t_bpass	Lenght of anti-aliasing band-pass filters	64
n_freq_bpass	Number of frequency bands of anti-aliasing filters	8
oversamp_bpass	Oversampling rate of output of anti-aliasing filters	1
min_freq_bpass	Minimum frequency for anti-aliasing band-pass filters (Hz)	0
max_freq_bpass	Maximum frequency for anti-aliasing band-pass filters (Hz)	Nyquist
n_threads	Number of desired threads	1
max_memory	Maximum available memory in bytes	128 Mb

DERIVED TYPES FOR SEISMIC DATA

Derived type: seis3d_geom_type

Data structure that describes a seismic data set; used for both the input and output data. The data geometry can be either *irregularly sampled* or *regularly sampled*.

```
type seis3d_geom_type
  type(irr_trace_geom_type), pointer :: irr
  type(reg_trace_geom_type), pointer :: reg
end type seis3d_geom_type
```

Derived-types variables		
Name	*Description*	*Remarks*
`irr`	Pointer to description of irregular geometry	Allocated only if needed
`reg`	Pointer to description of regular geometry	Allocated only if needed

Derived type: irr_trace_geom_type

Data structure that contains the geometry information for a *irregularly sampled* data space.

```
type irr_trace_geom_type
  character(len=128)  :: t_unit,geom_unit
  integer :: n_dim,n_tr_max,n_tr,n_t,tot_n_coord
  real :: t_0,d_t
  real, pointer, dimension (:,:) :: sou
  real, pointer, dimension (:,:) :: rec
  real, pointer, dimension (:,:) :: cmp
  real, pointer, dimension (:,:) :: off
  real, pointer, dimension (:) :: azim
  real, pointer, dimension (:) :: aoff
end type irr_trace_geom_type
```

Scalar variables		
Name	*Description*	*Remarks*
`t_unit`	Physical units of the time axis	
`geom_unit`	Physical units of the geometry axes	
`n_dim`	Number of dimensions of the geometry	$1 \leq$ `n_dim` ≤ 3
`n_tr_max`	Maximum number of traces	
`n_tr`	Number of actual traces different from zero	$1 \leq$ `n_tr` $\leq$ `n_tr_max`
`n_t, t_0, d_t`	Time axis parameters	
`tot_n_coord`	Total number of coordinates	Used internally

Array variables		
Name	*Description*	*Shape*
`sou`	Source coordinates	`(1:n_tr,1:n_dim)`
`rec`	Receiver coordinates	`(1:n_tr,1:n_dim)`
`cmp`	CMP coordinates	`(1:n_tr,1:n_dim)`
`off`	Signed **full** offset	`(1:n_tr,1:n_dim)`
`azim`	Azimuth in radians	`(1:n_tr)`
`aoff`	Absolute **full** offset	`(1:n_tr)`

Derived type: reg_trace_geom_type

Data structure that contains the geometry information for a *regularly sampled* data space.

```
type reg_trace_geom_type
  character(len=128)  :: tz_unit,geom_unit
  character(len=128)  :: label_tz,label_cmpx,label_cmpy,label_aoff,label_azim
  integer :: n_tz,n_cmpx,n_cmpy,n_aoff,n_azim
  real :: tz_0,cmpx_0,cmpy_0,aoff_0,azim_0
  real :: d_tz,d_cmpx,d_cmpy,d_aoff,d_azim
end type reg_trace_geom_type
```

Scalar variables	
Name	*Description*
`tz_unit`	Physical units of the time/depth axis
`geom_unit`	Physical units of the geometry axes
`n_tz, tz_0, d_tz, label_tz`	Time/depth axis parameters
`n_cmpx, cmpx_0, d_cmpx, label_cmpx`	CMP x-axis parameters
`n_cmpy, cmpy_0, d_cmpy, label_cmpy`	CMP y-axis parameters
`n_aoff, aoff_0, d_aoff, label_aoff`	Absolute-offset axis parameters
`n_azim, azim_0, d_azim, label_azim`	Azimuth axis parameters

DERIVED TYPES FOR OPERATOR STATUS AND MEMORY REQUIREMENTS

Derived type: genkir3d_info_type

This data structure is returned by the `amp_amp` functions. When the `want_info` variable is `.true.` the data structure must be filled with information regarding the memory requirements of the function. In particular, it returns a constant equal to the the memory requirements that are independent of the data sizes, and a set of proportionality constants realted to the data sizes. If the operator definition function does not require additional memory, the structure can be simply initialized with zeros by setting `info=info_default`, where `info_default` is a properly initialized parameter structure.

When the `want_info` variable is `.false.` only the `status` must be set. For successful completion `status` is set equal to 0.

```
type genkir3d_info_type
  integer :: status
  integer :: mem_fixed
  integer :: mem_n_t_inp,mem_n_tz_out
  integer :: mem_n_tr_inp,mem_n_tr_out
end type  genkir3d_info_type
```

Scalar variables	
Name	*Description*
`status`	Return the status of the call. Set to 0 if successful
`mem_fixed`	Memory requirement independent from data sizes
`mem_n_t_inp`	Proportionality constant for length of input time axis
`mem_n_tz_out`	Proportionality constant for length of output time/depth axis
`mem_n_tr_inp`	Proportionality constant for number of input traces
`mem_n_tr_out`	Proportionality constant for number of output traces

INTERFACES

To use effectively the Genkir3D package the user needs to know the interfaces of the different components. There are three kind of components:

1. Main module
2. Operator definition
3. Other user-definable components

INTERFACES FOR MAIN MODULE

The Genkir3D package is simply invoked by calling the `genkir3d` procedure. Therefore the understanding of the interface of this procedure is crucial. This section describes the interface of `genkir3d` and of its initialization function `genkir3d_init`.

Interface: genkir3d

The procedure `genkir3d` has many parameters, the most of which are optionals. The complete list of arguments of `genkir3d` is:

```
subroutine genkir3d(param,out_geom, out_geom_block, &
                    trace_interp_real,trace_interp_complex, &
                    spat_interp_real, spat_interp_complex, &
                    map_amp, map_amp_pre, map_amp_post, &
                    oper_domain)
type(genkir3d_param_type), intent (in) :: param
type(seis3d_geom_type), optional,  intent (in) :: out_geom,out_geom_block
external trace_interp_real,trace_interp_complex
external spat_interp_real,spat_interp_complex
external map_amp,map_amp_pre,map_amp_post
external oper_domain
optional trace_interp_real,trace_interp_complex
optional spat_interp_real,spat_interp_complex
optional map_amp,map_amp_pre,map_amp_post
optional oper_domain
```

Derived-type arguments		
Name	*Description*	*Remarks*
`param`	Processing parameter structure	
`out_geom`	Geometry of whole output space	Spraying or gathering
`out_geom_block`	Geometry of output space block	Spraying or gathering

External procedures	
Name	*Description*
`trace_interp_real`	Real-trace interpolation
`trace_interp_complex`	Complex-trace interpolation
`spat_interp_real`	Real spatial interpolation
`spat_interp_complex`	Complex spatial interpolation
`map_amp`	Operator transformation
`map_amp_pre`	Trace-to-trace transformation applied *before* `map_amp`
`map_amp_post`	Trace-to-trace transformation applied *after* `map_amp`
`oper_domain`	Operator domain

Interface: genkir3d_init

This routine initialize the parameters stored in `param` with sensible defaults. If the `verb` flag is `.true.` it prints to the screen these defaults.

```
subroutine genkir3d_init(verb,param)
logical :: verb
type(genkir3d_param_type), intent (inout) :: param
```

Scalar arguments	
Name	*Description*
`verb`	Verbose flag. Print defaults to screen if `.true.`

Derived-type arguments	
Name	*Description*
`param`	Processing parameter structure

INTERFACES FOR OPERATOR-DEFINITION MODULES

Interface: map_amp

The `map_amp` function computes the transformation from `n_tr_inp` input traces into `n_tr_out` output traces. The transformation is defined by the five arrays `map`, `amp`, `first_samp`, `last_samp`, and `max_freq`.

The `map` array is indexed in the ouput time/depth sampling, and contains the corresponding time coordinates of the input trace. These input-time coordinates are normalized so to be in samples, and not physical units.

The `amp` array is indexed in the ouput time/depth sampling, and contains the amplitude scaling factor of the transformation.

The `first_samp` and `last_samp` arrays define the valid range of the transformation, as the first and last output samples.

The `max_freq` array is indexed in the ouput time/depth sampling, and contains the high cut-off frequency of the bandpass filter applied to the input trace. The maximum frequency must be expressed as fraction of the Nyquist frequency.

The shapes of the output arrays, and whether each input array is allocated and initialized, depend on the spatial mapping method that is used to apply the operator.

- **Trace-to-trace**
  - `n_tr_inp`=1
  - `n_tr_out`=1
  - `max_freq`: not allocated
  - `inp_cent_coord`: not allocated
  - `inp_geom%reg`: not allocated
  - `inp_geom%irr`: initialized by Genkir3D
  - `out_geom%reg`: not allocated
  - `out_geom%irr`: initialized by Genkir3D
- **Gathering**
  - `n_tr_inp`=n_tr_inp
  - `n_tr_out`=1

  - `max_freq`: allocated
  - `inp_cent_coord`: not allocated
  - `inp_geom%reg`: not allocated
  - `inp_geom%irr`: initialized by Genkir3D
  - `out_geom%reg`: not allocated
  - `out_geom%irr`: initialized by Genkir3D

- **Spraying**
  - `n_tr_inp`=1
  - `n_tr_out`=(2*nh_z1+1)*(2*nh_z2+1)
  - `max_freq`: allocated
  - `inp_cent_coord`=initialized by `oper_domain`
  - `inp_geom%reg`: not allocated
  - `inp_geom%irr`: initialized by Genkir3D
  - `out_geom%reg`: initialized by Genkir3D
  - `out_geom%irr`: initialized by `oper_domain`

```
function map_amp (want_info,inv_map,inv_amp,inp_geom,out_geom, &
      first_samp, last_samp, map, amp, inp_cent_coord) result (info)

type(genkir3d_info_type) :: info
logical, intent(in)      :: want_info,inv_map,inv_amp
type (seis3d_geom_type), intent(in) :: inp_geom
type (seis3d_geom_type), intent(in) :: out_geom
integer, dimension(:,:), intent(out) :: first_samp,last_samp
real,  dimension (:,:,:), intent(out)  :: map,amp,max_freq
real, pointer,  optional,  dimension (:,:,:)  :: inp_cent_coord
```

Scalar arguments	
Name	*Description*
`want_info`	If `.true.` return memory requirements in `info`
`inv_map`	Compute mapping of inverse operator
`inv_amp`	Compute amplitude of inverse operator

Derived-type arguments	
Name	*Description*
`info`	Information on operator
`inp_geom`	Geometry of input traces
`out_geom`	Geometry of output traces

Array arguments		
Name	*Description*	*Shape*
`first_samp`	First valid output sample	(1:n_tr_inp,1:n_tr_out)
`last_samp`	Last valid output sample	(1:n_tr_inp,1:n_tr_out)
`map`	Mapping function	(1:n_tz_out,1:n_tr_inp,1:n_tr_out)
`amp`	Amplitude function	(1:n_tz_out,1:n_tr_inp,1:n_tr_out)
`max_freq`	Maximum frequency	(1:n_tz_out,1:n_tr_inp,1:n_tr_out)
`inp_cent_coord`	Input centered coordinates	(-nh_z1:nh_z1,-nh_z2:nh_z2,2)

Interface: oper_domain

The `oper_domain` function computes the operator domain for **spraying**-like operators. The operator is applied in two steps: first by spraying the input traces into an *intermediate* output space. Then, the intermediate traces are summed into the *real* output space.

The operator domain is defined in the intermediate output space. This intermediate output space is centered around the input traces, and is usually the most convenient to define the operator itself. The coordinates of the intermediate traces in the intermediate space are stored in the `inp_cent_coord` array. The *operator aperture* is defined by the shape of the array `inp_cent_coord`.

When the mapping between the intermediate space and the real output space is time-invariant, the **midpoint** coordinates of the intermediate traces in the real output space are stored in the `out_geom%irr` structure. When this mapping is time-variant, the midpoint coordinates in the real output space, of each sample in the intermediate traces, are stored in the `out_coord` array. The information on the ouput mapping is used by the spatial interpolator within Genkir3D.

```
function oper_domain (inv_map,inv_amp,inp_geom,out_geom, &
                      inp_cent_coord, out_coord) result (stat)

integer                :: stat
logical, intent(in)    :: inv_map,inv_amp
type (seis3d_geom_type), intent(in) :: inp_geom
type (seis3d_geom_type), intent(inout) :: out_geom
real, pointer,  dimension (:,:,:)  :: inp_cent_coord
real, pointer,  dimension (:,:,:)  :: out_coord
```

Scalar arguments	
Name	*Description*
`stat`	Status of the call. Set to 0 if successful
`inv_map`	Compute mapping of inverse operator
`inv_amp`	Compute amplitude of inverse operator

Derived-type arguments	
Name	*Description*
`inp_geom`	Geometry of input traces
`out_geom`	Geometry of output traces

Array arguments		
Name	*Description*	*Shape*
`inp_cent_coord`	Input centered coordinates	(-nh_z1:nh_z1,-nh_z2:nh_z2,2)
`out_coord`	Map from intermediate samples to output samples	(n_tz_out, (2*nh_z1+1)*(2*nh_z2+1),2)

INTERFACES FOR OTHER USER-DEFINABLE COMPONENTS

Interface: trace_interp_real and trace_interp_complex

These functions perform the time-axis transformation defined by the four arrays `map`, `amp`, `first_samp`, `last_samp`, and `max_freq`, that are initialized by the `map_amp` function. The function `trace_interp_real` is used for real traces, while the function `trace_interp_complex` is used for complex traces.

```
function trace_interp_real ( &
   adj, first_samp, last_samp, map, amp, inp, out) result (stat)
integer                 :: stat
logical, intent (in)    :: adj
integer, dimension(:,:) :: first_samp,last_samp
real,  dimension (:,:,:)  :: map,amp
real,  dimension (:,:)  :: inp, out

function trace_interp_complex ( &
   adj, first_samp, last_samp, map, amp, inp, out) result (stat)
integer                 :: stat
logical, intent (in)    :: adj
integer, dimension(:,:) :: first_samp,last_samp
complex,  dimension (:,:,:)  :: map,amp
complex,  dimension (:,:)  :: inp, out
```

Scalar arguments	
Name	*Description*
`stat`	Status of the call. Set to 0 if successful
`adj`	*Pull* interpolation (`adj=.false.`) vs. *push* interpolation (`adj=.true.`)

Array arguments		
Name	*Description*	*Shape*
`first_samp`	First valid output sample	(1:n_tr_inp,1:n_tr_out)
`last_samp`	Last valid output sample	(1:n_tr_inp,1:n_tr_out)
`map`	Mapping function	(1:n_tz_out,1:n_tr_inp,1:n_tr_out)
`amp`	Amplitude function	(1:n_tz_out,1:n_tr_inp,1:n_tr_out)
`max_freq`	Maximum frequency	(1:n_tz_out,1:n_tr_inp,1:n_tr_out)
`inp_bpass`	Bandpassed input traces	(1:n_t_inp*oversamp_bpass,1:n_tr_inp)
`out`	Output traces	(1:n_tz_out,1:n_tr_out)

Interface: spat_interp_real and spat_interp_complex

These functions perform a spatial interpolation between a *regularly* sampled data space and an *irregularly* sampled one. When the parameter `adj=.true.` the irregular data is sprayed into the regular one. On the contrary, when the parameter `adj=.false.` the irregular data is reconstructed by interpolating the regular data. The function `spat_interp_real` is used for real traces, while the function `spat_interp_complex` is used for complex traces.

These interpolation functions are needed by Genkir3D to regrid the intermediate output space into the real output space, when the spraying mapping is applied.

The mapping from the intermediate space into the real output space is defined either by only the structure `out_geom%irr` or by both `out_geom%irr` and `out_coord`. When the mapping between the intermediate space and the real output space is time-invariant, the **midpoint** coordinates of the intermediate traces in the real output space are stored in the `out_geom%irr` structure. When this mapping is time-variant, the midpoint coordinates in the real output space, of each sample in the intermediate traces, are stored in the `out_coord` array.

```
function spat_interp_real (adj, irr_geom, reg_geom, irr_data, reg_data, &
                           out_coord)  result (stat)
integer                 :: stat
logical, intent (in)    :: adj
type(irr_trace_geom_type), pointer :: irr_geom
type(reg_trace_geom_type), pointer :: reg_geom
real,  dimension (:,:)  :: irr_data
real,  dimension (:,:,:,:,:)  :: reg_data
real, pointer,  dimension (:,:,:)  :: out_coord

function spat_interp_complex (adj, irr_geom, reg_geom, irr_data, reg_data, &
                              out_coord) result (stat)
integer                 :: stat
logical, intent (in)    :: adj
type(irr_trace_geom_type), pointer :: irr_geom
type(reg_trace_geom_type), pointer :: reg_geom
complex,  dimension (:,:)  :: irr_data
complex,  dimension (:,:,:,:,:)  :: reg_data
```

```
real, pointer,  dimension (:,:,:)  :: out_coord
```

Scalar arguments	
Name	*Description*
`stat`	Status of the call. Set to 0 if successful
`adj`	*Pull* interpolation from regular to irregular (`adj=.false.`) *Push* interpolation from irregular to regular (`adj=.true.`)

Derived-type arguments	
Name	*Description*
`irr_geom`	Geometry of irregular data
`reg_geom`	Geometry of regular data

Array arguments		
Name	*Description*	*Shape*
`irr_data`	Irregularly sampled traces	(n_t_inp,n_tr_inp)
`reg_data`	Regularly sampled traces	(n_tz_out,n_cmpx,n_cmpy,n_aoff,n_azim) if #1 (n_tz_out,n_cmpx,n_aoff,n_azim,n_cmpy) if #2
`out_coord`	Map from irregular to regular	(n_t_inp,n_tr_inp,2)

- #1 if `out_sailing_lines = .false.` in `spat_interp_init`
- #2 if `out_sailing_lines = .true.` in `spat_interp_init`

INCLUDED MODULES

Included in the Genkir3D package there are some widely used modules that are available to the user without the need of coding them. This section describes these supported modules and the parameters needed to properly initialize them.

All the supported modules belong to one of the following categories:

1. Operator definition
2. Trace interpolation
3. Spatial interpolation

OPERATOR-DEFINITION MODULES

As mentioned above the operator are defined by a transformation and on operator domain. The following transformation and operator domain modules are included in Genkir3D. In some cases (e.g. DMO) the definition of transformation and the definition of the operator domain are closely linked In other cases they are not

Copy & window

This module defines the copy of the input trace into the output trace. The output trace can be a subset of the inpt trace if the output sampling rate is an integer multiple of the input sampling rate, and/or the output first sample is later than the input first sample. It requires `spat_map=trace_to_trace` in the parameters.

- `map_amp_init = none`
- `map_amp = window_map_amp`
- `oper_domain = none`

Stacking

This module defines a the operator domain for stacking, that is the input trace. It requires `spat_map=spray` in the parameters.

- `map_amp_init =  none`
- `map_amp = none`
- `oper_domain = point_domain`

Constant velocity NMO

This module defines a constant-velocity NMO-transformation of the input trace into the output trace. It requires `spat_map=trace_to_trace` in the parameters.

- `map_amp_init =  mov3d_hyper_const_init`
- `map_amp = mov3d_hyper_const`
- `oper_domain = none`

The interface of `mov3d_hyper_const_init` is:

```
function mov3d_hyper_const_init (slow_nmo) result (stat)
real :: slow_nmo
integer :: stat
```

Variable velocity NMO

This module defines a velocity NMO-transformation of the input trace into the output trace. It requires `spat_map=trace_to_trace` in the parameters.

- `map_amp_init = init_nmo_params`
- `map_amp = nmo_map_amp`
- `oper_domain = none`

The interface of `mov3d_hyper_const_init` is:

```
subroutine init_nmo_params(n1_in,o1_in,d1_in)
integer,intent(in) :: n1_in
real,intent(in) :: o1_in,d1_in
```

Constant velocity DMO

This module defines a constant-velocity DMO-transformation of the input trace into a set of output traces. It requires `spat_map=spray` in the parameters.

- `map_amp_init = dmo3d_map_amp_init`
- `map_amp = dmo3d_map_amp`
- `oper_domain = dmo3d_oper_domain`

The interface of `dmo3d_map_amp_init` is:

```
function dmo3d_map_amp_init (max_slow_param,oversamp_domain) result (stat)
real, optional :: oversamp_domain
real :: max_slow_param
integer :: stat
```

Constant velocity 3-D prestack migration

This module defines a constant-velocity prestack migration operator. It requires `spat_map=gather` in the parameters.

- `map_amp_init = mig3d_const_init`
- `map_amp = mig3d_const`

The interface of `mig3d_const_init` is:

```
function mig3d_const_init (slow_mig) result (stat)
real :: slow_mig
integer :: stat
```

TRACE-INTERPOLATION MODULES

Linear trace interpolation

This module linearly interpolates the time axis of the input trace according to the defined mapping and amplitude functions and sum the results into the output trace. The input frequency bands are interpolated by nearest-neighborhood interpolation.

- `trace_interp_real = trace_interp_lin_real`
- `trace_interp_complex = trace_interp_lin_complex`

Nearest-neighborhood trace interpolation

This module interpolates the tiem axis of the input trace with a nearest-neighborhood interpolator, according to the defined mapping and amplitude functions and sum the results into the output trace. The input frequency bands are interpolated by nearest-neighborhood interpolation.

- `trace_interp_real = trace_interp_near_real`
- `trace_interp_complex = trace_interp_near_complex`

Linear/Nearest-neighborhood trace interpolation

This module interpolates the time axis of the input trace with a nearest-neighborhood interpolator, according to the defined mapping and amplitude functions and sum the results into the output trace. The input frequency bands are interpolated by linear interpolation.

- `trace_interp_real = trace_interp_near_lin_freq_real`
- `trace_interp_complex = trace_interp_near_lin_freq_complex`

SPATIAL-INTERPOLATION MODULES

Linear spatial interpolation

This module spatially interpolates the input traces into the output space by a bilinear interpolator in the midpoint space and a nearest-neighborhood interpolator along the offset and azimuth axes.

- `spat_interp_real = spat_interp_lin_near_real`
- `spat_interp_complex = spat_interp_lin_near_complex`

The interface of `spat_interp_lin_near_real_init` and `spat_interp_lin_near_complex_init` are the same. For example the interface of `spat_interp_lin_near_real_init` is

```
function spat_interp_near_near_real_init (out_sailing_lines) result (stat)
integer                 :: stat
logical, optional       :: out_sailing_lines
```

The parameter `out_sailing_lines` is defaulted to `.false.` when is not present, or the initialization function is not called.

Nearest-neighborhood spatial interpolation

This module spatially interpolates the input traces into the output space by a nearest-neighborhood interpolator in the midpoint space and a nearest-neighborhood interpolator along the offset and azimuth axes.

- `spat_interp_real = spat_interp_near_near_real`
- `spat_interp_complex = spat_interp_near_near_complex`

The interface of `spat_interp_near_near_real_init` and `spat_interp_near_near_complex_init` are the same as in for the corresponding linear interpolators. For example the interface of `spat_interp_lin_near_real_init` is

```
function spat_interp_near_near_real_init (out_sailing_lines) result (stat)
integer                 :: stat
logical, optional       :: out_sailing_lines
```

The parameter `out_sailing_lines` is defaulted to `.false.` when is not present, or the initialization function is not called.

EXAMPLES

There are four kinds of examples:

1. Main programs
2. Parameter handling routines
3. Operator-definition modules
4. Makefiles

Examples of main programs

Main program for 3-D DMO

```
!
!       Example for Dmo3d using GenKir3D
!
!%
program Examp

use dmo3d_types

use genkir3d_mod
use dmo3d_mod

implicit none

interface
        subroutine param_dmo (dmo3d_param,gen_param,out_geom)
          type(dmo3d_param_type) :: dmo3d_param
          type(genkir3d_param_type) :: gen_param
          type(seis3d_geom_type) :: out_geom
        end subroutine param_dmo
end interface

type(dmo3d_param_type) :: dmo3d_param
type(genkir3d_param_type) :: gen_param
type(seis3d_geom_type) :: out_geom

integer :: ierr=0

call initpar()
call init_3d()

call genkir3d_init(.false.,gen_param)

call param_dmo (dmo3d_param,gen_param, out_geom)

if(dmo3d_map_amp_init (dmo3d_param%max_slow) /= 0 ) then
```

```
  call seperr('Problems with initializing dmo3d_map_amp_init')
end if

call genkir3d(param=gen_param,  &
 out_geom=out_geom, &
 trace_interp_real=trace_interp_lin_real, &
 spat_interp_real=spat_interp_lin_near_real, &
 map_amp=dmo3d_map_amp,  &
 oper_domain=dmo3d_domain)

! clean-up
if(allocated(out_geom%reg)) deallocate(out_geom%reg,stat=ierr)
if(ierr .ne. 0) call seperr('Troubles in deallocating out_geom%reg')
if(allocated(out_geom%irr)) deallocate(out_geom%irr,stat=ierr)
if(ierr .ne. 0) call seperr('Troubles in deallocating out_geom%irr')

stop

end program Examp
```

Examples of parameter handling

Parameter handling routine for 3-D DMO

```
subroutine param_dmo (dmo3d_param,gen_param,out_geom)

use dmo3d_types
use seis3d_types
use genkir3d_types

implicit none

integer, intrinsic ::  mp_numthreads

type(dmo3d_param_type) :: dmo3d_param
type(genkir3d_param_type), intent (inout) :: gen_param
type(seis3d_geom_type), intent (inout) :: out_geom


logical reg_out
integer getch,auxpar,esize,ione,izero,ierr
integer n_threads_env
real rzero,rone,maxreal

izero=0
ione=1
rzero=0.
rone=1.
maxreal=huge(maxreal)

! Getting specific Test parameters for param
```

```
from par:       logical reg_out=.true.

! Getting specific dmo3d parameters for param
from par:       character trace_interp_type:dmo3d_param%trace_interp_type='linear'
from par:       character chain_maps:dmo3d_param%chain_maps='none'
from par:       real max_slow:dmo3d_param%max_slow=maxreal
from par:       real slow_nmo:dmo3d_param%slow_nmo=0.


! Getting parameters for param
from par:       logical inv_map:gen_param%inv_map=gen_param%inv_map
from par:       logical inv_amp:gen_param%inv_amp=gen_param%inv_amp
from par:       logical adj:gen_param%adj=gen_param%adj
from par:       logical add:gen_param%add=gen_param%add
from par:       logical verb:gen_param%verb=gen_param%verb
from par:       logical debug1:gen_param%debug1=gen_param%debug1
from par:       logical debug2:gen_param%debug2=gen_param%debug2
from par:       logical pipe_inp:gen_param%pipe_inp=gen_param%pipe_inp
from par:       logical pipe_out:gen_param%pipe_out=gen_param%pipe_out
from par:       logical seq_inp_acc:gen_param%seq_inp_acc=gen_param%seq_inp_acc
from par:       logical seq_out_acc:gen_param%seq_out_acc=gen_param%seq_out_acc
from par:       logical out_in_lines:gen_param%out_in_lines=gen_param%out_in_lines

from par:       character tag_inp:gen_param%tag_inp=trim(gen_param%tag_inp)

from par:       character tag_out:gen_param%tag_out=trim(gen_param%tag_out)

from par:       character inp_type:gen_param%inp_type=trim(gen_param%inp_type)
from par:       character out_type:gen_param%out_type=trim(gen_param%out_type)

from par:       integer n_threads:gen_param%n_threads=mp_numthreads()
from par:       integer max_memory:gen_param%max_memory=gen_param%max_memory

! Getting parameters for res
from par:       integer n_t:gen_param%n_t=gen_param%n_t
from par:       real t_0:gen_param%t_0=gen_param%t_0
from par:       real d_t:gen_param%d_t=gen_param%d_t

if(reg_out) then
  allocate(out_geom%reg,stat=ierr)
  if(ierr .ne. 0) call seperr('Troubles in allocating out_geom%reg')

  from par:     character tz_unit:out_geom%reg%tz_unit='s'
  from par:     character geom_unit:out_geom%reg%geom_unit='m'
  from par:     character label_tz:out_geom%reg%label_tz='Time('//trim(out_geom%reg%tz_unit)//')'

  from par:     integer n_tz:out_geom%reg%n_tz=gen_param%n_t
  from par:     real tz_0:out_geom%reg%tz_0=gen_param%t_0
  from par:     real d_tz:out_geom%reg%d_tz=gen_param%d_t

  from par:     character label_cmpx:out_geom%reg%label_cmpx='Cmpx('//trim(out_geom%reg%geom_unit)//
  from par:     character label_cmpy:out_geom%reg%label_cmpy='Cmpy('//trim(out_geom%reg%geom_unit)//
  from par:     character label_aoff:out_geom%reg%label_aoff='Offset('//trim(out_geom%reg%geom_unit)
  from par:     character label_azim:out_geom%reg%label_azim='Azimuth(deg)'
```

```
  from par:     integer n_cmpx:out_geom%reg%n_cmpx
  from par:     integer n_cmpy:out_geom%reg%n_cmpy
  from par:     integer n_aoff:out_geom%reg%n_aoff=ione
  from par:     integer n_azim:out_geom%reg%n_azim=ione
  from par:     real cmpx_0:out_geom%reg%cmpx_0
  from par:     real cmpy_0:out_geom%reg%cmpy_0
  from par:     real aoff_0:out_geom%reg%aoff_0=rzero
  from par:     real azim_0:out_geom%reg%azim_0=rzero
  from par:     real d_cmpx:out_geom%reg%d_cmpx
  from par:     real d_cmpy:out_geom%reg%d_cmpy
  from par:     real d_aoff:out_geom%reg%d_aoff=maxreal
  from par:     real d_azim:out_geom%reg%d_azim=360

end if

! checking parameters
n_threads_env=mp_numthreads()
if(n_threads_env /= gen_param%n_threads) then
  write(0,*)'Reset number of threads'
  write(0,*)'because default threads',n_threads_env, &
  'different from requested = ',gen_param%n_threads
  gen_param%n_threads=n_threads_env
end if

return
end
```

Examples of operator-definition modules

Operator definition for constant-velocity NMO

```
function mov3d_hyper_const (inv_map,inv_amp,inp_geom,out_geom, &
      first_samp, last_samp, map, amp, inp_cent_coord) result (stat)
logical                 :: inv_map,inv_amp
integer                 :: stat
type (seis3d_geom_type), intent(in) :: inp_geom
type (seis3d_geom_type), intent(in) :: out_geom
integer, dimension(:,:), intent(out) :: first_samp,last_samp
real,  dimension (:,:,:), intent(out)  :: map,amp
real, pointer,  optional,  dimension (:,:,:)  :: inp_cent_coord

...


if(.not. inv_map) then
  in_bound = .true.
  moveout= slow_sq*aoff_sq
  if(moveout < 0.) then
    first_samp(1,1) = &
    max(1,ceiling(((t_0_dat + sqrt(-moveout))-t_0_mod)*inv_d_t_mod)+1)
  else
    first_samp(1,1)=1
```

```
  end if
  last_samp(1,1)=n_t_mod
  i_samp_mod = first_samp(1,1)
  do while ((i_samp_mod < n_t_mod ) .and. in_bound)
    t_mod=t_0_mod + (i_samp_mod-1)*d_t_mod
    t_dat=sqrt(t_mod*t_mod + moveout)
    map(i_samp_mod,1,1)=(t_dat-t_0_dat)*inv_d_t_dat + 1
    if(inv_amp) then
      if(t_mod > 0.) then
        amp(i_samp_mod,1,1)=t_dat/t_mod
      else
        amp(i_samp_mod,1,1)=1.
      end if
    else
      if(t_dat > 0.) then
        amp(i_samp_mod,1,1)=t_mod/t_dat
      else
        amp(i_samp_mod,1,1)=1.
      end if
    end if
    if(map(i_samp_mod,1,1) > n_t_dat) then
      in_bound=.false.
      last_samp(1,1)=i_samp_mod-1
    end if
    i_samp_mod=i_samp_mod + 1
  end do
else
  call seperr('mov3d_hyper_const: cannot implement inv_map with NMO')
end if
```

Operator definition for 3-D DMO

Following is an example of implementing the operator domain definition function for 3-D DMO:

```
function dmo3d_domain (inv_map,inv_amp,inp_geom,out_geom, &
                       inp_cent_coord,out_coord) result (stat)
logical                 :: inv_map,inv_amp
integer                 :: stat
type (seis3d_geom_type), intent(in) :: inp_geom
type (seis3d_geom_type), intent(inout) :: out_geom

real, pointer,  dimension (:,:,:)  :: inp_cent_coord
real, pointer,  dimension (:,:,:)  :: out_coord

! no cross-line extension for operator
nh_z2=0

sin_inp=sin(inp_geom%irr%azim(1))
cos_inp=cos(inp_geom%irr%azim(1))
h_inp=.5*inp_geom%irr%aoff(1)
```

```
h_cos_inp=h_inp*cos_inp
h_sin_inp=h_inp*sin_inp

! Operator aperture limit according to min velocity
t_bar=(2.*h_inp*max_slow)/inp_geom%irr%d_t

! Estimate sampling rate along z1 and corresponding number of samples
d_z1 = sqrt((cos_inp*out_geom%reg%d_cmpx)**2  + &
            (sin_inp*out_geom%reg%d_cmpy)**2) / h_inp
nh_z1=(1./d_z1)-1

n_z1=1+2*nh_z1
n_z2=1+2*nh_z2
n_z1z2=n_z1*n_z2
n_dim=inp_geom%irr%n_dim

! allocate array for operator domain
allocate(inp_cent_coord(-nh_z1:nh_z1,-nh_z2:nh_z2,2), stat=ierr)
if(ierr .ne. 0) call seperr('point_domain: Troubles in allocating cordinate arrays')

! allocate arrays for output geometry info n
allocate( out_geom%irr%cmp(n_z1z2,inp_geom%irr%n_dim), stat=ierr)
if(ierr .ne. 0) call seperr('point_domain: Troubles in allocating output geometry arrays')

! loop over operator domain
do i_z2=-nh_z2,nh_z2
  do i_z1=-nh_z1,nh_z1
    z1 = i_z1*d_z1
! computer input_centered coordinates
    inp_cent_coord(i_z1,i_z2,1) = z1
    inp_cent_coord(i_z1,i_z2,2) = 0.
    i_z1z2=(i_z1+nh_z1)+(i_z2+nh_z2)*n_z1+1
! compute corresonding output space coordinates
    out_geom%irr%cmp(i_z1z2,1) = inp_geom%irr%cmp(1,1) + z1 * h_cos_inp
    out_geom%irr%cmp(i_z1z2,2) = inp_geom%irr%cmp(1,2) + z1 * h_sin_inp
  end do
end do

stat=0

return
end function dmo3d_domain
```

Following is an example of implementing the transformation function for 3-D DMO:

```
function dmo3d_map_amp (inv_map,inv_amp,inp_geom,out_geom, &
      first_samp, last_samp, map, amp, inp_cent_coord) result (stat)
logical                 :: inv_map,inv_amp
integer                 :: stat
type (seis3d_geom_type), intent(in) :: inp_geom
type (seis3d_geom_type), intent(in) :: out_geom
integer, dimension(:,:), intent(out) :: first_samp,last_samp
```

```
real,  dimension (:,:,:), intent(out)  :: map,amp
real, pointer,  optional,  dimension (:,:,:)  :: inp_cent_coord

logical in_bound
integer :: i_samp_out,i_tr_out,i_z1
real aoff_sq,t_out,t_inp,inv_d_t_inp,rat,z1,ratsq
real inv_d_t_out,t0_out
real ::  epsi,max_t_inp,t0_inp

if(.not. present(inp_cent_coord)) then
  call seperr('dmo3d_map_amp: inp_cent_coord should be present')
end if

epsi=epsilon(z1)

if(.not. inv_map) then

inv_d_t_inp = 1./inp_geom%irr%d_t
t0_inp = inv_d_t_inp * inp_geom%irr%t_0
! loop over operator domain
  do i_z1=lbound(inp_cent_coord,dim=1),ubound(inp_cent_coord,dim=1)

    i_tr_out=i_z1-lbound(inp_cent_coord,dim=1)+1

! compute stretching factor
    z1=inp_cent_coord(i_z1,0,1)
    ratsq=1./(1. - z1*z1)
    rat=sqrt(ratsq)

! set first output sample
    first_samp(1,i_tr_out)=ceiling(t0_inp/rat)

! set input time limits according to aperture limitation
    if(abs(z1) > epsi) then
      max_t_inp=min(inp_geom%irr%n_t,(t_bar/abs(z1))-t0_inp)
    else
      max_t_inp=inp_geom%irr%n_t
    end if

! set values before looping
    last_samp(1,i_tr_out)=out_geom%irr%n_t
    i_samp_out=0
    in_bound = .true.
 !loop over output samples
    do while ((i_samp_out < out_geom%irr%n_t) .and. in_bound)
      i_samp_out=i_samp_out+1
      t_out=out_geom%irr%t_0 + (i_samp_out-1)*out_geom%irr%d_t
! compute time map
      t_inp=t_out*rat
      map(i_samp_out,1,i_tr_out)=(t_inp - inp_geom%irr%t_0)*inv_d_t_inp +1
! compute amplitude
      if(inv_amp) then
        amp(i_samp_out,1,i_tr_out)=1.
      else
```

```
        amp(i_samp_out,1,i_tr_out)=(2.*ratsq-1.)
      end if
! check to be within input trace limits
      if(map(i_samp_out,1,i_tr_out) > max_t_inp) then
        in_bound=.false.
        last_samp(1,i_tr_out)=i_samp_out-1
      end if
    end do
  end do
else

inv_d_t_inp = 1./inp_geom%irr%d_t
inv_d_t_out = 1./out_geom%irr%d_t
t0_out = inv_d_t_out * out_geom%irr%t_0
! loop over operator domain
  do i_z1=lbound(inp_cent_coord,dim=1),ubound(inp_cent_coord,dim=1)

    i_tr_out=i_z1-lbound(inp_cent_coord,dim=1)+1

! compute stretching factor
    z1=inp_cent_coord(i_z1,0,1)
    ratsq=(1. - z1*z1)
    rat=sqrt(ratsq)

! set output output sample according to aperture limitation
    if(abs(z1) > epsi) then
      last_samp(1,i_tr_out)=min(out_geom%irr%n_t, (t_bar/abs(z1))-t0_out)
    else
      last_samp(1,i_tr_out)=out_geom%irr%n_t
    end if

! set values before looping
    first_samp(1,i_tr_out)=1
    i_samp_out=last_samp(1,i_tr_out)
    in_bound = .true.
 !loop over output samples
    do while ((i_samp_out > 1 ) .and. in_bound)
      i_samp_out=i_samp_out - 1
      t_out=out_geom%irr%t_0 + (i_samp_out-1)*out_geom%irr%d_t
! compute time map
      t_inp=t_out*rat
      map(i_samp_out,1,i_tr_out)=(t_inp - inp_geom%irr%t_0)*inv_d_t_inp +1
! compute amplitude
      if(inv_amp) then
        amp(i_samp_out,1,i_tr_out)=1.
      else
        amp(i_samp_out,1,i_tr_out)=(2.*ratsq-1.)
      end if
! check to be within input trace limits
      if(map(i_samp_out,1,i_tr_out) < 1.) then
        in_bound=.false.
        first_samp(1,i_tr_out)=i_samp_out + 1
      end if
    end do
```

```
  end do
end if

stat=0

return
end function dmo3d_map_amp
```

Operator definition for constant-velocity 3-D prestack migration

Following is an example of implementing the transformation function for 3-D prestack migration with constant velocity:

```
function mig3d_const (want_info,inv_map,inv_amp,inp_geom,out_geom, &
      first_samp, last_samp, max_freq, map, amp, inp_cent_coord) result (info)
type(genkir3d_info_type) :: info
logical, intent(in)      :: want_info,inv_map,inv_amp
type (seis3d_geom_type), intent(in) :: inp_geom
type (seis3d_geom_type), intent(in) :: out_geom
integer, dimension(:,:), intent(out) :: first_samp,last_samp
real,  dimension (:,:,:), intent(out)  :: map,amp,max_freq
real, pointer,  optional,  dimension (:,:,:)  :: inp_cent_coord

logical in_bound
integer :: n_t_mod,n_t_dat,i_samp_mod,n_tr_mod,n_tr_dat,i_tr_dat
integer :: n_dim,i_dim
real ::  t_mod,t_mod_sq,t_dat,inv_d_t_dat,inv_d_t_mod
real ::  t_0_mod,t_0_dat,d_t_mod,dpos,dneg
real :: dm(3)

if(.not. want_info) then
!{

n_dim=out_geom%irr%n_dim
n_tr_mod=out_geom%irr%n_tr
n_t_mod=out_geom%irr%n_t
t_0_mod=out_geom%irr%t_0
d_t_mod=out_geom%irr%d_t
inv_d_t_mod=1./out_geom%irr%d_t

n_tr_dat=inp_geom%irr%n_tr
n_t_dat=inp_geom%irr%n_t
t_0_dat=inp_geom%irr%t_0
inv_d_t_dat=1./inp_geom%irr%d_t

max_freq=1.

if(.not. inv_map) then
!{
  do i_tr_dat=1,n_tr_dat
  !{
```

```
    dpos=0.
    dneg=0.
    do i_dim=1,n_dim
      dm(i_dim)=(out_geom%irr%cmp(1,i_dim) - &
                 inp_geom%irr%cmp(i_tr_dat,i_dim))
      dpos = dpos + (dm(i_dim) + &
                   .5*inp_geom%irr%off(i_tr_dat,i_dim))**2
      dneg = dneg + (dm(i_dim) - &
                   .5*inp_geom%irr%off(i_tr_dat,i_dim))**2
    end do
    dpos=dpos*slow_sq
    dneg=dneg*slow_sq

    in_bound = .true.
    first_samp(i_tr_dat,1)=1
    last_samp(i_tr_dat,1)=n_t_mod
    i_samp_mod = first_samp(i_tr_dat,1)
    do while ((i_samp_mod < n_t_mod ) .and. in_bound)
      t_mod=.5*(t_0_mod + (i_samp_mod-1)*d_t_mod)
      t_mod_sq=t_mod**2
      t_dat=sqrt(t_mod_sq + dpos) + sqrt(t_mod_sq + dneg)
      map(i_samp_mod,i_tr_dat,1)=(t_dat-t_0_dat)*inv_d_t_dat + 1

      if(inv_amp) then
        if(t_mod_sq > 0.) then
          amp(i_samp_mod,i_tr_dat,1)=t_dat/t_mod
        else
          amp(i_samp_mod,i_tr_dat,1)=1.
        end if
      else
        if(t_dat > 0.) then
          amp(i_samp_mod,i_tr_dat,1)=t_mod/t_dat
        else
          amp(i_samp_mod,i_tr_dat,1)=1.
        end if
      end if

      if(map(i_samp_mod,i_tr_dat,1) > n_t_dat) then
        in_bound=.false.
        last_samp(i_tr_dat,1)=i_samp_mod-1
      end if
      i_samp_mod=i_samp_mod + 1
    end do
  !}
  end do
!}
else
!{
  call seperr('mig3d_const: cannot implement inv_map with mig3d')
!}
end if

  info=info_default
```

```
  info%status=0
!}
else
!{
 info=info_default
!}
end if

return
end function mig3d_const
```

.

Examples of makefiles

Example of Makefile

```
ifndef SEPINC
SEPINC = /usr/local/src/our/sep4/config
endif
include ${SEPINC}/SEP.distr.top

BINDIR = /biondo/bin/$(SEP_ARCH)
INCDIR = /biondo/include/genkir3d_v1/$(SEP_ARCH)
LIBDIR = /biondo/lib/genkir3d_v1/$(SEP_ARCH)

LOCAL := Examp

UF90FLAGS= -I${INCDIR}
UF90LIBS= ${SEPLIB3DF90} ${SEPLIBF90} ${SEPLIB3D}  ${SEPLIBF} ${SEPLIB}
UF90LIBS= ${LIBDIR}/libgenkir3d.a -lsepf90 -lsep3df90  -lsep3d  \
        -lsepconvertf90 -lsepconvert -lsepauxf90 -lsepaux \
          ${SEPLIBF} ${SEPLIB} -lmp

all: $(ALL)
        @echo "Done making all Examp "

$(ALL): %: $(BINDIR)/%


deinstall:
        $(RM) $(strip $(patsubst %,$(BINDIR)/%,$(ALL) ))


EXAMPOBJS =  Examp.o  param_examp.o


$(BINDIR)/Examp: $(addprefix $(OBJDIR)/, $(EXAMPOBJS))
        ${F90_LN}
```

```
include ${SEPINC}/SEP.distr.bottom
```