Chapter 12

SOFTWARE SUPPORT

"**Ratfor**" (RATional FORtran) is a dialect of Fortran that is more concise than raw Fortran. Our present Ratfor "compiler," `ratfor90`, is a simple word-processing program (written[1] in Perl and freely distributed) that inputs an attractive Fortran-like dialect and outputs Fortran90. Mainly, the word-processor produces Fortran statements like `end do`, `end if`, `end program` and `end module`, from the Ratfor "}". Ratfor source is about 25-30% smaller than the equivalent Fortran, so it is equivalently more readable.

Bare-bones **Fortran** is our most universal computer language for computational physics. For general programming, however, it has been surpassed by **C**. Ratfor is Fortran with C-like syntax. Ratfor was invented by the people[2] who invented C. After inventing C, they realized that they had made a mistake (too many semicolons) and they fixed it in Ratfor, although it was too late for C. Otherwise, Ratfor uses C-like syntax, the syntax that is also found in the popular languages **C++** and **Java**.

At SEP we supplemented Ratfor77 by preprocessors to give Fortran77 the ability to allocate memory on the fly. These abilities are built into Fortran90 and are seamlessly included in **Ratfor90**. To take advantage of Fortran90's new features while maintaining the concise coding style provided by Ratfor, we had to write a new Ratfor preprocessor, Ratfor90, which produces Fortran90 rather than Fortran77 code.

You should be able to read Ratfor if you already know Fortran or any similar computer language. Writing Ratfor is easy if you already know Fortran because written Fortran is valid Ratfor. You can mix Ratfor and Fortran. The Ratfor processor is not a compiler but a simple word-processing program which passes Fortran (which it does not understand) through unchanged. The Ratfor processor converts the Ratfor dialect to Fortran. To maximize the amount of Ratfor, you will need to know its rules. Here they are:

Statements on a line may be separated by ";". Statements may be grouped together with braces { }. Do loops do not require statement numbers because { } defines the range. Given that `if( )` is true, the statements in the following { } are done. `else{ }` does what you expect.

[1] The Ratfor90 preprocessor was written by my colleague, Bob Clapp.
[2] Kernighan, B.W. and Plauger, P.J., 1976, Software Tools: Addison-Wesley.

We may *not* contract `else if` to `elseif`. We may omit the braces { } where they contain only one statement. `break` (equivalent to the Fortran90 `exit`) causes premature termination of the enclosing { }. `while( ) { }` repeats the statements in { } while the condition ( ) is true. Ratfor recognizes `repeat { ... } until( )` as a loop that tests at the bottom. `next` causes skipping to the end of any loop and a retrial of the test condition. `next` (equivalent to the Fortran90 `cycle` statement) is rarely used, and the Ratfor90 coder may write either `next` or `cycle`. Here we encounter an inconsistency between Fortran and C-language. Where Ratfor uses `next`, the C-language uses `continue` (which in Ratfor and Fortran is merely a place holder for labels). The Fortran relational operators `.gt.`, `.ge.`, `.ne.`, etc. may be written >, >=, !=, etc. The logical operators `.and.` and `.or.` may be written && and ||. Anything from a # to the end of the line is a comment. A line may be continued in Ratfor by ending it with the underscore charactor "_" (like Fortran90's &).

Indentation in Ratfor is used for readability. It is not part of the Ratfor language. Choose your own style. I have overcondensed. There are two **pitfall**s associated with indentation. The beginner's pitfall is to assume that a `do` loop ends where the indentation ends. The loop actually ends after the first statement. A larger scope for the `do` loop is made by enclosing multiple statements in braces. The other pitfall arises in any construction like `if() ... if() ... else`. The `else` goes with the last `if()` regardless of indentation. If you want the `else` with the earlier `if()`, you must use braces like `if() { if() ... } else ...`. Ratfor also recognizes the looping statement used in C, C++, and Java. It is `for(`initialize`;` condition`;` reinitialize`) { }`.

12.0.1 Changes and backward compatibility

We were forced to make one change to Ratfor90 because of new things in Fortran90. Ratfor77 allows & and | for the logical operators && and ||. While attractive, it is not part of the C family of languages and we had to drop it because Fortran90 adopts & for line continuation.

Because we are not compiler writers, we dropped a rarely used feature of Ratfor77 that was not easy for us to implement and is ugly anyway: Ratfor77 recognizes `break 2` which escapes from {{ }}.

Changing all the code that generated illustrations for four textbooks (of various ages) also turned up a few more issues: Fortran90 uses the words `scale` and `matmul` as intrinsics. Old Fortran77 programs using those words as variable names must be changed. Ratfor77 unwisely allowed variables of intrinsic (undeclared) types. We no longer allow this. Ratfor90 forces `implicit none`.

New features in Ratfor90 are bracketed type, subroutine, function, and module procedures. In some ways this a further step towards the C, C++, Java model. It makes complicated modules, subroutines inside subroutines, and other advanced features of Fortran90 easier to interpret. Ratfor90 has better error messages than Ratfor77. Besides the use of stderr, a new file (`ratfor_problem`) marks the difficulty.

12.0.2 Examples

Below are simple Ratfor subroutines for erasing an array (`zero()`); (`null()`); for copying one array to another (`copy()`); for the signum function $sgn(x) = x/|x|$ (`signum()`); and (`tcaf`), a program using fortran90 modules and overloading to transient convolution.

12.0.3 Memory allocation in subroutines

For backward compatibility we allow the "temporary" memory allocation introduced by our Ratfor77 processor for example:

```
temporary real*4 data(n1,n2,n3), convolution(j+k-1)
```

These declarations must follow other declarations and precede the executable statements. Automatic arrays are supported in Fortran90. To allow full code compatibility, `Ratfor90` simply translates this statement to

```
real*4 data(n1,n2,n3), convolution(j+k-1).
```

12.0.4 The main program environment

`Ratfor90` includes some traditional SEP local-memory-allocation and data-base-I/O statements that are described below. It calls an essential seplib initialization routine `initpar()`, organizes the self-doc, and simplifies data-cube input. The basic syntax for memory allocation is `allocate:  real x(n1,n2)`. `Ratfor90` translates this syntax into a call to dynamically allocate a `allocatable` array. See the on-line self-documentation or the manual pages for full details. Following is a complete Ratfor program for a simple task:

```
# <in.H  Scale scaleval=1. > out.H
#
#       Copy input to output and scale by scaleval
# keyword generic scale
#%
integer n1, n2, n3, esize
from history:   integer n1, n2, n3, esize
if (esize !=4)  call erexit('esize != 4')
allocate:       real x(n1,n2)
subroutine scaleit( n1,n2, x)
integer i1,i2, n1,n2
real    x(n1,n2), scaleval
from par:       real scaleval=1.
call hclose()                   # no more parameter handling.
call sreed('in', x, 4*n1*n2)
do i1=1,n1
        do i2=1,n2
                x(i1,i2) = x(i1,i2) * scaleval
call srite( 'out', x, 4*n1*n2)
return;        end
```

12.1 SERGEY'S MAIN PROGRAM DOCS

Many of the illustrations in this book are made with main programs that can be reused (in the SEP environment) for other applications. Here is a summary of their documentation.

12.1.1 Autocorr - compute autocorrelation for helix filters

```
Autocorr < filt.H  > autocorr.H
```

Reads a helix filter. Outputs the positive lag of its autocorrelation (no space wasted).

from/to history	**integer**	*n1*	filter size
	integer array	*lag*	comma-separated list of filter lags
	real	*a0*=1	zero-lag coefficient

Modules: *helix.r90*, *autocorr.r90*

12.1.2 Bin2 - nearest neighbor binning in 2-D

```
Bin2 < triplets.H > map.H
```

Bins (x,y,z) data triplets. Normalizes by bin fold.

from history	**integer**	*n1, n2*	*n1* is number of triplets, *n2* must be 3
from par	**integer**	*n1, n2* – map size	
	real	*o1, o2, d1, d2* – map dimensions	

Modules: *bin2.lop*

12.1.3 Conv - convolve two helix filters

```
Conv < filt1.H other=filt2.H > conv.H
```

Outputs the convolution of filt1 and filt2.

from/to history	**integer**	*n1*	filter size
	integer array	*lag*	comma-separated list of filter lags

Modules: *helix.r90*, *conv.r90*

12.1.4 Decon - Deconvolution (N-dimensional)

```
Decon < data.H filt= predictive=0 > decon.H
```

Deconvolution: predictive, Lomoplan, steep dip. Uses the helix and patching technology.

from history	**integer array**	*n*	*n1*, *n2*, *n3*, etc
from par	**filename**	*filt*	helix-type local PEF
	logical	*predictive*=0	predictive deconvolution
	integer	*rect1* (optional)	smoothing on the first axis
from aux (filt)	**integer**	*dim*	number of dimensions
	integer array	*w*	patch size
	integer array	*k*	number of windows

Modules: *tent.r90*, *patching.r90*, *loconvol.r90* , *helix.r90 triangle.r90*
See also: `Lopef`, `Helicon`

12.1.5 Devector - create one output filter from two input filters

```
Devector < filt1.H other=filt2.H > filt.H
```

Uses Wilson's factorization. filt = sqrt (filt1**2 + filt2**2)

from history and from aux (other)	**integer**	*n1*	number of filter coefficients
	integer arra	*n2,n3,...*	number of filters
	integer array	*lag*	helix filter lags
from par	**integer**	*niter*=20	number of WIlson's iterations
	integer	*n1*	number of output filter coefficients
	integer array	*lag*	output lags

Modules: *wilson.r90*, *autocorr.r90*, *compress.r90*

12.1.6 Helderiv - Helix derivative filter in 2-D

```
Helderiv < in.H helix=1 na=16 > out.H
```

Factors the laplacian operator. Applies helix derivative. Loops over n3

from history	**integer**	*n1*, *n2*	
from par	**logical**	*helix*=1	if 0, apply the gradient filter on the 1st axis
	integer	*na*=16	filter size (half the number of nonzero coefficients)
	real	*eps*=0.001	zero frequency shift on the laplacian

Modules: *helicon.lop*, *helderiv.r90*

12.1.7 Helicon - Helix convolution and deconvolution (N-dimensional!)

```
Helicon < in.H filt= adj=0 div=0 > out.H
```

Applies helix convolution (polynomial multiplication) or deconvolution (polynomial division). One is the exact inverse of the other. Watch for helical boundary conditions.

from history	**integer array**	*n*	reads *n1*, *n2*, *n3*, ...
from par	**filename**	*filt*	helix filter file
	integer	*adj*=0	apply adjoint (backward) filtering
	integer	*div*=0	apply inverse recursive filtering (polynomial division)
from aux (filt)	**integer array**	*h*	helix grid (can be *h1*, *h2*, ...)
	integer array	*lag*=1,...,n1	comma separated list of filter lags
	real	*a0*=1	zero-lag filter coefficient

Modules: *helicon.lop*, *polydiv.lop*, *regrid.r90*, *helix.r90*

12.1.8 Helocut - Helix Lowcut filter in 2-D

```
Helocut < in.H helix=1 na=16 eps=0.1 > out.H
```

Applies helix convolution with a low-cut factor, based on factoring the laplacian filter. Also loops over n3.

from history	**integer**	*n1*, *n2*	
from par	**logical**	*helix*=1	if 0, apply the gradient filter on the 1st axis
	real	*eps*	sets the lowcut frequency
	integer	*na*=16	filter size (half the number of nonzero coefficients)

Modules: *helicon.lop*, *helocut.r90*

12.1.9 Hole - Punch ellipsoidal hole in 2-D data

```
Hole < data.H > hole.H
```

Hole's dimensions and orientation are currently fixed

from history	**integer**	*n1, n2*

See also: `Make`

12.1.10 Igrad - take gradient on the first axis

```
Igrad < map.H > grad.H
```

Works on 2-D data, gradient is (1,-1) filter

from history	**integer**	*n1, n2*

Modules: *igrad1.lop*

12.1.11 LPad - Pad and interleave traces

```
LPad < small.H jump=2 mask= > large.H
```

Each initial trace is followed by *jump* zero traces, the same for planes.

from history	**integer**	*n1, n2, n3*	
from par	**integer**	*jump*=2	how much to expand the axes
	filename	*mask*	selector for known traces (same size as output)
to history	**integer**	*n2*=n2*jump (if *n2* > 1), *n3*=n3*jump (if *n3* > 1)	

See also: `LPef`

12.1.12 LPef - Find PEF on aliased traces

```
LPef < in.H jump=2 a= center=1 gap=0 > out.H
```

Finds a prediction-error filter, assuming missing traces

from history	**integer array**	*n*	reads *n1*, *n2*, *n3*, etc.
from par	**integer**	*jump*=2	how much to expand the axes
	integer array	*a*=	PEF size
	integer array	*center*=1	PEF centering
	integer array	*gap*=0	PEF gapping

Modules: *lace.r90*, *helix.r90*, *print.r90*, *compress.r90*
See also: `Pef`

12.1.13 Lapfill2 - fill missing data by minimizing the Laplacian

```
Lapfill2 < map.H > filled.H
```

Works on 2-D data only.

from history	**integer**	*n1, n2*	
from par	**integer**	*niter*=200	number of CG iterations

Modules: *lapfill.r90*
See also: `Miss`, `MSMiss`

12.1.14 LoLPef - Find PEF on aliased traces (with patching)

```
LoLPef < in.H jump=2 a= center=1 gap=0 > out.H
```

Finds a prediction-error filter, assuming missing traces

from history	**integer array**	*n*	reads *n1*, *n2*, *n3*, etc.
from par	**integer array**	*w*=20,20,6	patch size
	integer array	*k* (optional)	number of windows
	integer	*jump*=2	how much to expand the axes
	integer array	*a*=	PEF size
	integer array	*center*=1	PEF centering
	integer array	*gap*=0	PEF gapping

Modules: *lolace.r90*
See also: `Pef`

12.1.15 Lomiss - Missing data interpolation with a prescribed helix filter

12.1.16 (in local patches)

```
Lomiss < in.H prec=1 niter=100 filt= [mask=] > interp.H
```

Fills missing data by mimimizing the data power after convolution. Works in any number of dimensions!

from history	**integer**	*n1, n2, n3*	
from par	**integer**	*prec*=1	use preconditioning for missing data interpolation
	integer	*niter*=100	number of iterations
	filename	*filt*	helix filter
	filename	*mask* (optional)	selector for known data
from aux (sfilt, nfilt)	**integer**	*dim*	number of dimensions
	integer array	*w*	patch size
	integer array	*k*	number of windows

Modules: *lomis2.r90*, *helix.r90*, *tent.r90*

12.1.17 Lopef - Local Prediction-Error Filter (1-D, 2-D, and 3-D)

```
Lopef < data.H dim=3 steepdip=0 > pef.H
```

Local prediction-error filters are estimated with the helix and patching technology. Can also find filters for steep-dip deconvolution. Currently works in 1, 2, and 3 dimensions.

from history	**integer**	*n1, n2, n3*	
	real	*d1, d2, d3* (for steep-dip decon)	
from par	**integer**	*dim*=3	number of dimensions
	integer array	*w*=20,20,6	patch size
	integer array	*a*=5,2,1	filter size
	integer array	*k* (optional)	number of windows
	integer array	*gap*=0,0,0	filter gap
	integer array	*ctr* (optional)	filter centering
	logical	*steepdip*=0	steep-dip decon PEF
	real	*vel*=1.7	velocity for steep-dip decon
	real	*tgap*=0.03	time gap for steep-dip decon
	filename	*mask* (optional)	data selector

Modules: *bound.r90*, *steepdip.r90*, *shape.r90*, *lopef.r90*, *print.r90*, *helix.r90*
See also: `Pef`, `Decon`

12.1.18 Losignoi - Local signal and noise separation (N-dimensional)

```
Losignoi < data.H sfilt= nfilt= eps= > sign.H
```

Signal and noise separation by inversion (super-deconvolution). Uses the helix and patching technologies.

from history	**integer array**	*n*	*n1*, *n2*, *n3*< etc
from par	**filename**	*sfilt*, *nfilt*	helix-type signal and noise local PEF
	real	*eps*	the magic scaling parameter
	integer	*niter*=20	number of iterations
from aux (sfilt, nfilt)	**integer**	*dim*	number of dimensions
	integer array	*w*	patch size
	integer array	*k*	number of windows

Modules: *tent.r90*, *patching.r90*, *signoi.r90* , *helix.r90*
See also: `Decon`, `Lopef`, `Helicon`

12.1.19 MSHelicon - Multi-scale Helix convolution (N-dimensional!)

```
Helicon < in.H filt= ns= jump= adj=0 > out.H
```

Applies multiscale helix convolution.

from history	**integer array**	*n*	reads *n1*, *n2*, *n3*, ...
from par	**filename**	*filt*	helix filter file
	integer	*adj*=0	apply adjoint (backward) filtering
	integer	*ns*	number of scales
	integer array	*jump*=0	filter scales
from aux (filt)	**integer array**	*h*	helix grid (can be *h1*, *h2*, ...)
	integer array	*lag*=1,...,n1	comma separated list of filter lags
	real	*a0*=1	zero-lag filter coefficient

Modules: *mshelicon.lop*, *regrid.r90*, *mshelix.r90*

12.1.20 MSMiss - Multiscale missing data interpolation (N-dimensional)

```
MSMiss < in.H prec=1 niter=100 filt= [mask=] > interp.H
```

Fills missing data by mimimizing the data power after convolution.

from history	**integer array**	*n*	reads *n1*, *n2*, *n3*, ...
from aux (filt)	**integer**	*ns*	number of scales
	integer array	*jump*	comma separated list of scales, e.g. 1,2,4
from par	**integer**	*prec*=1	use preconditioning for missing data interpolation
	integer	*niter*=100	number of iterations
	filename	*filt*	helix filter
	filename	*mask* (optional)	selector for known data

Modules: *msmis2.r90*, *mshelix.r90*, *bound.r90*

12.1.21 MSPef - Multi-scale PEF estimation

```
MSPef < in.H a= center= gap=0 ns= jump= [maskin=] [maskout=] > pef.H
```

Estimates a multi-scale PEF. Works in N dimensions

from history	**integer array**	*n*	reads *n1*, *n2*, *n3*
from par	**integer array**	*a*=	PEF size
	integer	*niter*=2*prod(a) (optional)	number of PEF iterations
	integer array	*center*	PEF centering
	integer array	*gap*=0	PEF gapping
	integer	*ns*	number of scales
	integer array	*jump*	comma separated list of scales, e.g. 1,2,4
	filename	*maskin, maskout* (optional)	data selectors

Modules: *mspef.r90*, *misinput.r90*, *mshelix.r90 createmshelixmod.r90*, *print.r90*
See also: `MSMiss Pef`

12.1.22 Make - generate simple 2-D synthetics with crossing plane waves

```
Make n1=100 n2=14 n3=1 n=3 p=3 t1=4 t2=4 > synth.H
```

Plane waves have fixed slopes, but random amplitudes

from par	**integer**	*n1*=100, *n2*=14, *n3*=1	data size
	integer	*n*=3	slope
	integer	*p*=3	power for generating random distribution
	integer	*t1*=3, *t2*=3	width of trinalge smoother on the two waves

Modules: *triangle.lop*, *random.f90* (for compatibility with Fortran-77)
See also: `Hole`

12.1.23 Minphase - create minimum-phase filters

```
Minphase < filt.H niter=20 > minphase.H
```

Uses Wilson's factorization. The phase information is lost.

from history	**integer**	*n1*	number of filter coefficients
	integer array	*n2,n3,...*	number of filters
	integer array	*lag*	helix filter lags
from par	**integer**	*niter*=20	number of WIlson's iterations

Modules: *wilson.r90*, *autocorr.r90*

12.1.24 Miss - Missing data interpolation with a prescribed helix filter

```
Miss < in.H prec=1 niter=100 padin=0 padout=0 filt= [mask=] > interp.H
```

Fills missing data by mimimizing the data power after convolution. Works in any number of dimensions!

from history	**integer**	*n1, n2, n3*	
from par	**integer**	*prec*=1	use preconditioning for missing data interpolation
	integer	*niter*=100	number of iterations
	integer	*padin*=0	pad data beginning
	integer	*padout*=0	pad data end
	filename	*filt*	helix filter
	filename	*mask* (optional)	selector for known data

Modules: *mis2.r90*, *bound.r90*, *helix.r90*

12.1.25 NHelicon - Non-stationary helix convolution and deconvolution

```
Helicon < in.H filt= adj=0 div=0 > out.H
```

Applies helix convolution (polynomial multiplication) or deconvolution (polynomial division). One is the exact inverse of the other. Watch for helical boundary conditions.

from history	**integer array**	*n*	reads *n1*, *n2*, *n3*, ...
from par	**filename**	*filt*	helix filter file
	integer	*adj*=0	apply adjoint (backward) filtering
	integer	*div*=0	apply inverse recursive filtering (polynomial division)
from aux (filt)	**integer array**	*h*	helix grid (can be *h1*, *h2*, ...)
	integer array	*lag*=1,...,n1	comma separated list of filter lags
	real	*a0*=1	zero-lag filter coefficient

Modules: *nhelicon.lop*, *npolydiv.lop*, *nhelix.r90*, *helix.r90*, *regrid.r90*

12.1.26 NPef - Estimate Non-stationary PEF in N dimensions

```
Pef < data.H a= center=1 gap=0 [maskin=] [maskout=] > pef.H
```

Estimates PEF by least squares, using helix convolution. Can ignore missing data

from history	**integer array**	*n*	reads *n1*, *n2*, *n3*, etc.
from par	**integer**	*niter*=100	number of iterations
	real	*epsilon*=0.01	regularization parameter
	integer array	*a*=	filter size
	integer array	*center*=1	zero-lag position (filter centering)
	integer array	*gap*=0	filter gap
	filename	*maskin, maskout* (optional)	data selectors
to history	**integer array**	*lag*	comma separated list of filter lags

Modules: *nhelix.r90*, *createnhelixmod.r90*, *nmisinput.r90*, *npef.r90*,
See also: `MSPef`, `Pef`, `NHelicon`

12.1.27 Nozero - Read (x,y,z) data triples, throw out values of z > thresh, transpose

```
Nozero < triplets.H thresh=-210 > transp.H
```

The program is tuned for the Sea of Galilee data set

from history	**integer**	*n1, n2*	*n2* is the number of triples, *n1* must equal 3
	real	*thresh*=-210 - threshold (default is tuned for Galilee)	
to history	**integer**	*n1, n2*	*n1* is the number of triples such that z < thresh
	n2=3		

See also: `Bin2`

12.1.28 Parcel - Patching illustration

```
Parcel < in.H w= k= > out.H
```

Transforms data to patches and back without the weighting compenssation.

	integer array	*w*	window size
	integer array	*k*	number of windows in different directions

Modules: *parcel.lop*, *cartesian.r90*

12.1.29 Pef - Estimate PEF in N dimensions

```
Pef < data.H a= [center=] [gap=] [maskin=] [maskout=] > pef.H
```

Estimates PEF by least squares, using helix convolution. Can ignore missing data

from history	**integer array**	*n*	reads *n1*, *n2*, *n3*, etc.
from par	**integer array**	*a*=	filter size
	integer	*niter*=2*prod(a) (optional)	number of
	PEF iterations		
	integer array	*center*=a/2+1 (optional)	zero-lag position (filter centering)
	integer array	*gap*=0 (optional)	filter gap
	filename	*maskin, maskout* (optional)	data selectors
to history	**integer array**	*lag*	comma separated list of filter lags

Modules: *shape.r90*, *bound.r90*, *misinput.r90*, *pef.r90*, *compress.r90*, *print.r90*, *helix.r90*
See also: `MSPef`, `Fillmiss`, `Helicon`, `Decon`

12.1.30 Sigmoid - generate sigmoid reflectivity model

```
Sigmoid n1=400 n2=100 o1=0 d1=.004 o2=0 d2=.032 > synth.H
```

Sigmoid reflectivity model in 2-D: complex geological structure.

from par	**integer**	*n1*=400, *n2*=100	data size
	integer	*large*=5*n1	layering size
	real	*o1*=0, *d1*=0.004, *o2*=0., *d2*=0.032	grid spacing

Modules: *random.f90* (for compatibility with Fortran-77)
See also: `Make`

12.1.31 Signoi - Local signal and noise separation (N-dimensional)

```
Signoi < data.H sfilt= nfilt= epsilon= > sig+noi.H
```

Signal and noise separation by optimization.

from history	**integer array**	*n*	*n1*, *n2*, *n3*< etc
from par	**filename**	*sfilt*, *nfilt*	helix-type signal and noise local PEF
	real	*eps*	the magic scaling parameter
	integer	*niter*=20	number of iterations

Modules: *signoi.r90* , *regrid.r90*
See also: `Losignoi`, `Pef`

12.1.32 Tentwt - Tent weight for patching

```
Tentwt dim=2 n= w= windwt= >wallwt.H
```

Computes the tent weight for patching.

from par	**integer**	*dim*=2	number of dimensions
	integer array	*n*	data size (n1, n2, etc)
	integer array	*w*	window size
	integer array	*k* (optional)	number of windows in different directions
	integer array	*a* (optional)	window offset
	integer array	*center* (optional)	window centering

Modules: *tent.r90*, *wallwt.r90*

12.1.33 Vrms2int - convert RMS velocity to interval velocity

```
Vrms2int < vrms.H weight= vrms= niter= eps= > vint.H
```

Least-square inversion, preconditioned by integration.

from history	**integer**	*n1, n2*	
from par	**integer**	*niter*	number of iterations
	real	*eps*	scaling for preconditioning
	filename	*weight*	data weight for inversion
	filename	*vrms*	predicted RMS velocity

Modules: *vrms2int.r90*

12.1.34 Wilson - Wilson's factorization for helix filters

```
Wilson < filt.H  niter=20 [n1= lag=] > minphase.H
```

Reads a helix autocorrelation (positive side of it). Outputs its minimum-phase factor.

from/to history	**integer**	*n1*	filter size
	integer array	*lag*	comma-separated list of filter lags
	real	*a0*=1	zero-lag coefficient
from par	**integer**	*niter*=20	number of Newton's iterations
	integer	*n1* (optional)	number of coefficients
	integer array	*lag* (optional)	comma-separated list of filter lags

Modules: *wilson.lop*, *helix.r90*, *compress.r90*

12.2 References

Claerbout, J., 1990, Introduction to `seplib` and SEP utility software: **SEP–70**, 413–436.

Claerbout, J., 1986, A canonical program library: **SEP–50**, 281–290.

Cole, S., and Dellinger, J., Vplot: SEP's plot language: SEP-**60**, 349–389.

Dellinger, J., 1989, Why does SEP still use Vplot?: SEP–**61**, 327–335.

Chapter 13

Entrance examination

1. (10 minutes) Given is a residual $\mathbf{r}$ where

$$\mathbf{r} \quad = \quad \mathbf{d}_0 - m_1 \mathbf{b}_1 - m_2 \mathbf{b}_2 - m_3 \mathbf{b}_3$$

   The data is $\mathbf{d}_0$. The fitting functions are the column vectors $\mathbf{b}_1$, $\mathbf{b}_2$, and $\mathbf{b}_3$, and the model parameters are the scalars m_1, m_2, and m_3. Suppose that m_1 and m_2 are already known. Derive a formula for finding m_3 that minimizes the residual length (squared) $\mathbf{r} \cdot \mathbf{r}$.

2. (10 minutes) Below is a subroutine written in a mysterious dialect of Fortran. Describe ALL the inputs required for this subroutine to multiply a vector times the *transpose* of a matrix.

```
# matrix multiply and its adjoint
#
subroutine matmult( adj, bb,         x,nx,   y,ny)
integer ix, iy,     adj,               nx,    ny
real                     bb(ny,nx), x(nx), y(ny)
if( adj == 0 )
        do iy= 1, ny
                y(iy) = 0.
else
        do ix= 1, nx
                x(ix) = 0.
do ix= 1, nx {
do iy= 1, ny {
        if( adj == 0 )
                        y(iy) = y(iy) + bb(iy,ix) * x(ix)
        else
                        x(ix) = x(ix) + bb(iy,ix) * y(iy)
        }}
return; end
```

Index