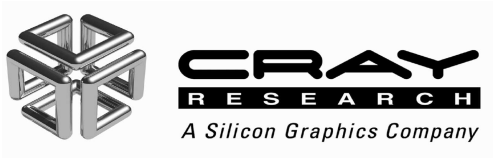
Migration of Common Focal Point Gathers

Jan Thorbecke*

Tuesday February 10, 1998



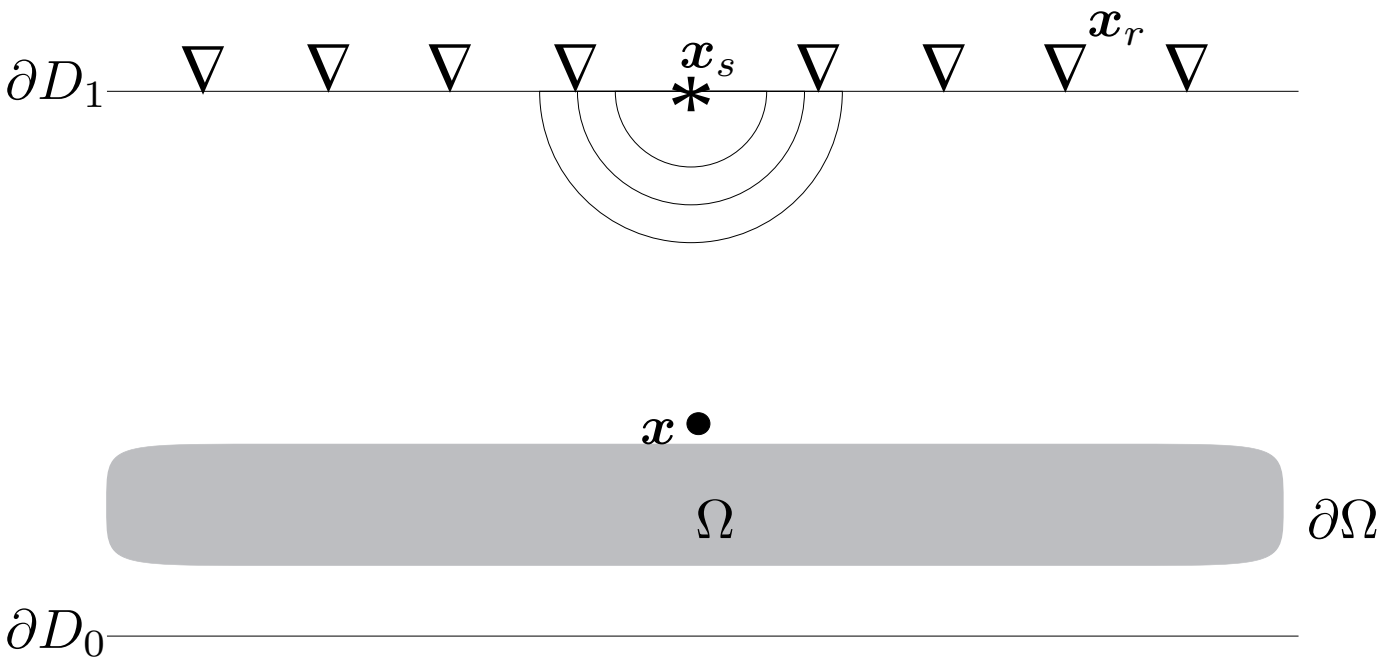
Part 1: CFP gathers

Part 2: Algorithm optimization

Part 3: CFP Migration examples

Focusing integral for receiver array

$$P^{-,s}(\boldsymbol{x},\boldsymbol{x}_s)=\int_{\partial D_1} W_p^{+,*}(\boldsymbol{x},\boldsymbol{x}_r)P^{-,s}(\boldsymbol{x}_r,\boldsymbol{x}_s)\mathrm{d}^2\boldsymbol{x}_r,$$



Focusing matrix for receiver array

Focusing result:

$$\tilde{\boldsymbol{P}}_i^-(z_f, z_s) = \tilde{\boldsymbol{F}}_i^-(z_f, z_r)\mathbf{P}(z_r, z_s)$$

with operator

$$\tilde{\boldsymbol{F}}_i^-(z_f, z_r) \approx \tilde{\boldsymbol{I}}_i^-(z_f)\left[\mathbf{W}^+(z_f, z_r)\right]^*$$

$$\tilde{\boldsymbol{F}}_i^-(z_f, z_r)\mathbf{W}^-(z_r, z_f) = \tilde{\boldsymbol{I}}_i^-(z_f)$$

and forward model

$$\mathbf{P}(z_r, z_s) = \mathbf{W}^-(z_r, z_f)\mathbf{R}^+(z_f)\mathbf{W}^+(z_f, z_s)\mathbf{S}(z_s)$$

gives

$$\tilde{\boldsymbol{P}}_i^-(z_f, z_s) = \tilde{\boldsymbol{I}}_i^-(z_f)\mathbf{R}^+(z_f)\mathbf{W}^+(z_f, z_s)\mathbf{S}(z_s)$$

Migration of CFP gathers

Focusing operator (source):

$$\tilde{\boldsymbol{F}}_i^-(z_f, z_r) \approx \tilde{\boldsymbol{I}}_i^-(z_f) \left[\mathbf{W}^+(z_f, z_r)\right]^*$$

Focusing result (response):

$$\tilde{\boldsymbol{P}}_i^-(z_f, z_s) = \mathbf{R}_i^+(z_f)\mathbf{W}^+(z_f, z_s)\mathbf{S}(z_s)$$

Migration steps

- Forward extrapolation of source with $\mathbf{W}^{+}(z_k, z_r) \rightarrow \tilde{\boldsymbol{F}}_i^{-}(z_k)$
- Inverse extrapolation of response with $\left[\mathbf{W}^{+}(z_k, z_s)\right]^{*} \rightarrow P_i^{-}(z_k)$
- Imaging at z_k with extrapolated source and response

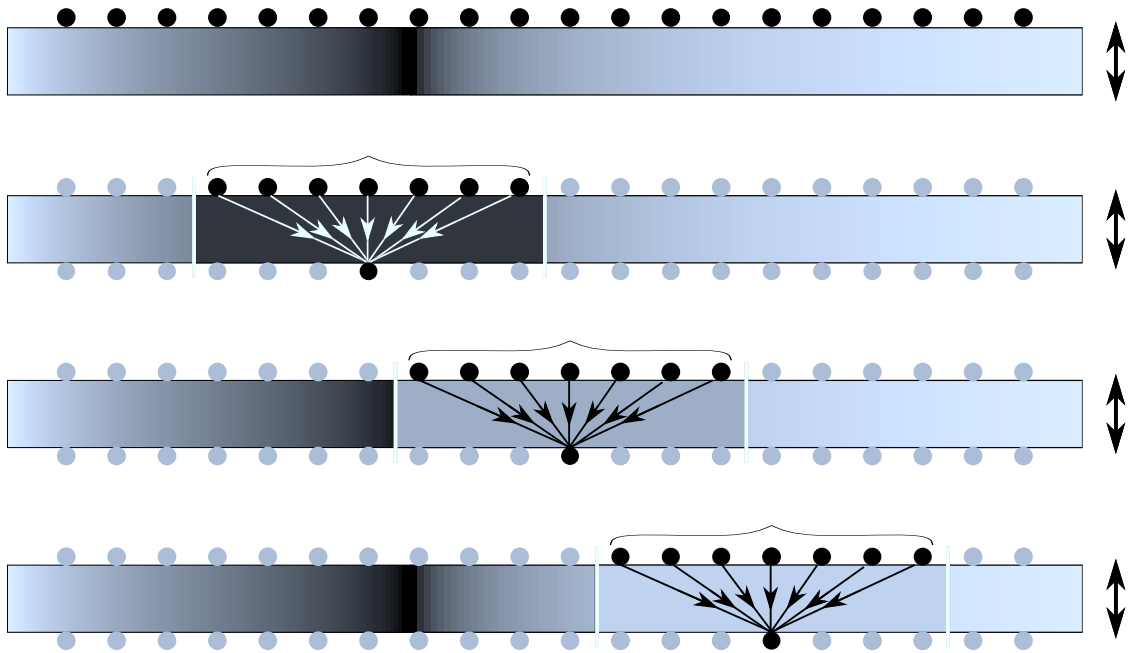
$$\mathbf{I}mage_j(z_k) = \sum_{\omega} \frac{F_{ij}^{-}(z_k)\left[P_{ij}^{-}(z_k)\right]^{*}}{F_{ij}^{+}(z_k)\left[F_{ij}^{-}(z_k)\right]^{*} + \epsilon}$$

Extrapolation

$$\tilde{\boldsymbol{P}}^{+}(z_k) = \mathbf{W}^{+}(z_k, z_r)\tilde{\boldsymbol{P}}^{+}(z_r)$$

$$\begin{bmatrix} \tilde{P}^{+}(0) \\ \vdots \\ \tilde{P}^{+}(j * \Delta x) \\ \vdots \\ \tilde{P}^{+}(N * \Delta x) \end{bmatrix} (z_k) = \Delta x \begin{bmatrix} W_{11} & \dots & W_{1j} & \dots & W_{1N} \\ \vdots & \ddots & \vdots & \dots & \vdots \\ W_{j1} & \dots & W_{jj} & \dots & W_{jN} \\ \vdots & \dots & \vdots & \ddots & \vdots \\ W_{N1} & \dots & W_{Nj} & \dots & W_{NN} \end{bmatrix} \begin{bmatrix} P^{+}(\Delta x) \\ \vdots \\ P^{+}(j * \Delta x) \\ \vdots \\ \tilde{P}^{+}(N * \Delta x) \end{bmatrix} (z_0)$$

Recursive Extrapolation



$$\tilde{\mathbf{P}}^{+}(z_k) = \mathbf{W}^{+}(z_k, z_r)\tilde{\mathbf{P}}^{+}(z_r)$$

Implementation in Blas # 1

Matrix-vector multiplication for general band matrix (Blas 2 cgbmv)

```
for (iom = iomin; iom <= iomax; iom++) {
    get P(z_0) and S(z_0);
    for (d = 1; d <= ndepth; d++) {
        calculate W from operator table
        call cgbmv( P(z_d), W*, P(z_0) );
        call cgbmv( S(z_d), W , S(z_0) );
        calculate Image(z_d);
        copy P(z_d) to P(z_0);
        copy S(z_d) to S(z_0);
    }
}
```

Total WallClock-time = **78.8** seconds.

Implementation in Blas

Using 'ssrun -fpcsamp' to determine bottle-necks:

98 % of the code is spend in the extrapolation loop of the code

60 % of the code is spend in cgbmv

37 % of the code is spend in constructing **W**

Performance relative to peak

Number of floating point operations with 'ssrun -ideal' and 'prof -op'

11577475914: Floating point operations (147 Mflops)

Note that the MIPS R10000 (195 MHz) is capable of doing an addition and a multiplication in one clock cycle, giving a peak performance of 390 Mflops.

Convolution loop #2

The convolution loop:

```
index1 = ix + hopl2;
for (j = 0; j < hopl; j++) {
    i1 = index1+j;
    wa += locdat[i1].r*opx[j];
    da += locsrc[i1].r*opx[j];
    i2 = index1-j;
    wa += locdat[i2].r*opx[j];
    da += locsrc[i2].r*opx[j];
}
```

Total WallClock-time = **39.8** seconds.

Compiling with 'cc -S' gives information about Software Pipelining

```
#<swps> Pipelined loop line 171 steady state
#<swps>
#<swps>   100 estimated iterations before pipelining
#<swps>       Not unrolled before pipelining
#<swps>    16 cycles per iteration
#<swps>    32 flops        (100% of peak) (madds count as 2)
#<swps>    16 flops        ( 50% of peak) (madds count as 1)
#<swps>    16 madds        (100% of peak)
#<swps>    10 mem refs     ( 62% of peak)
#<swps>    10 integer ops  ( 31% of peak)
#<swps>    36 instructions ( 56% of peak)
#<swps>     2 short trip threshold
#<swps>    15 integer registers used.
#<swps>    21 float registers used
```

11453823230: Floating point operations (288 Mflops)

A much higher number of flops, but is it the fastest code?

Note that the symmetry in the extrapolation operator is not used.

Symmetric Matrix-Vector multiplication #3

The convolution loop:

```
index1 = ix + hopl2;
index2 = lenx-index1-1;
for (j = 0; j < hopl; j++) {
    wa += (tmp1[index1+j] + tmp2[index2+j])*opx[j];
    da += (tmp3[index1+j] + tmp4[index2+j])*opx[j];
}
```

Total WallClock-time = **33.0** seconds.

Software Pipelining:

```
#<swps>   100 estimated iterations before pipelining
#<swps>       Not unrolled before pipelining
#<swps>    12 cycles per iteration
#<swps>    20 flops        ( 83% of peak) (madds count as 2)
#<swps>    12 flops        ( 50% of peak) (madds count as 1)
#<swps>     8 madds        ( 66% of peak)
#<swps>    10 mem refs     ( 83% of peak)
#<swps>     6 integer ops  ( 25% of peak)
#<swps>    28 instructions ( 58% of peak)
#<swps>     1 short trip threshold
#<swps>     9 integer registers used.
#<swps>    19 float registers used.
```

7329160832: Floating point operations (222 Mflops)

Less flops and faster code.

Symmetric operator # 4

The convolution loop;

```
index1 = ix + hopl2;
for (j = 0; j < hopl; j++) {
    i1 = index1+j;
    i2 = index1-j;
    wa += (locdat[i1]+locdat[i2])*opx[j];
    da += (locsrc[i1]+locsrc[i2])*opx[j];
}
```

Total WallClock-time = **31.6** seconds.

	Mflop	Time (s)	Mflop/s
Blas	11577	78.8	147
Madds	11453	39.8	288
Vector	7329	33.0	222
Symmetric	7328	31.6	232

Parallelization with directives

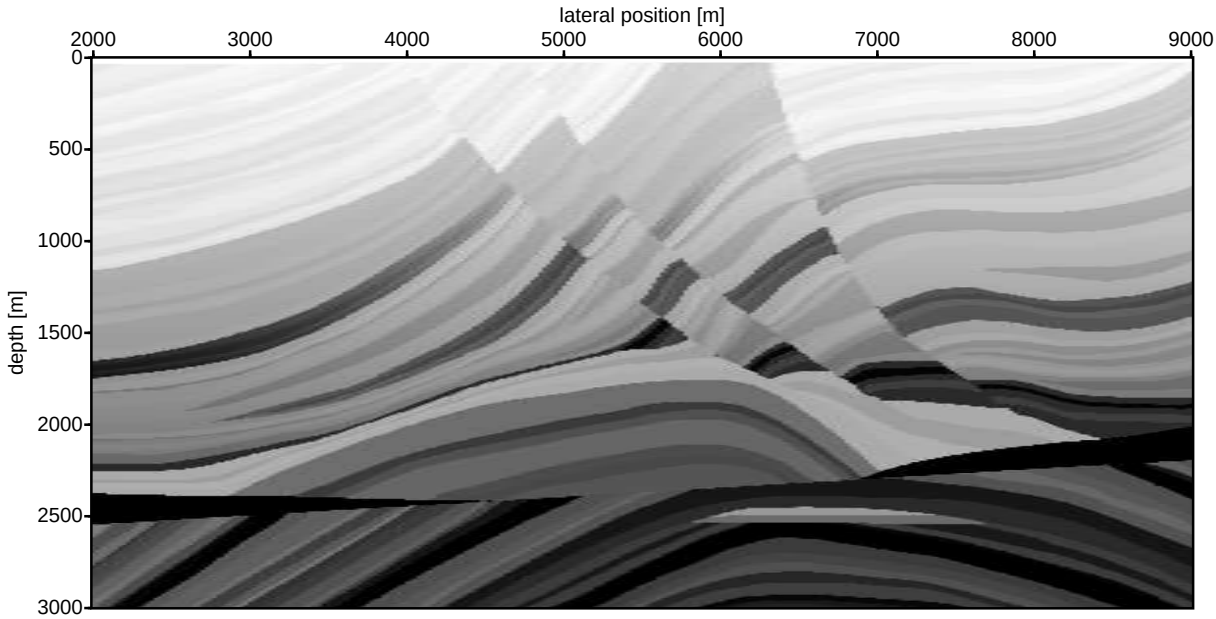
Around the frequency loop the compiler directives are inserted and nothing else is changed in the code.

```
#pragma parallel
#pragma shared(image)
#pragma byvalue(hopl, velmod, lenx, iomin, iomax, taper, dom)
#pragma local(iom, tmp1, tmp2, cprev, d)
        { /* start of parallel region */

 ........

/* start extrapolation for all frequencies, depths and x-positions */

#pragma pfor iterate(iom=iomin;iomax;1)


 } /* end of parallel region */
```

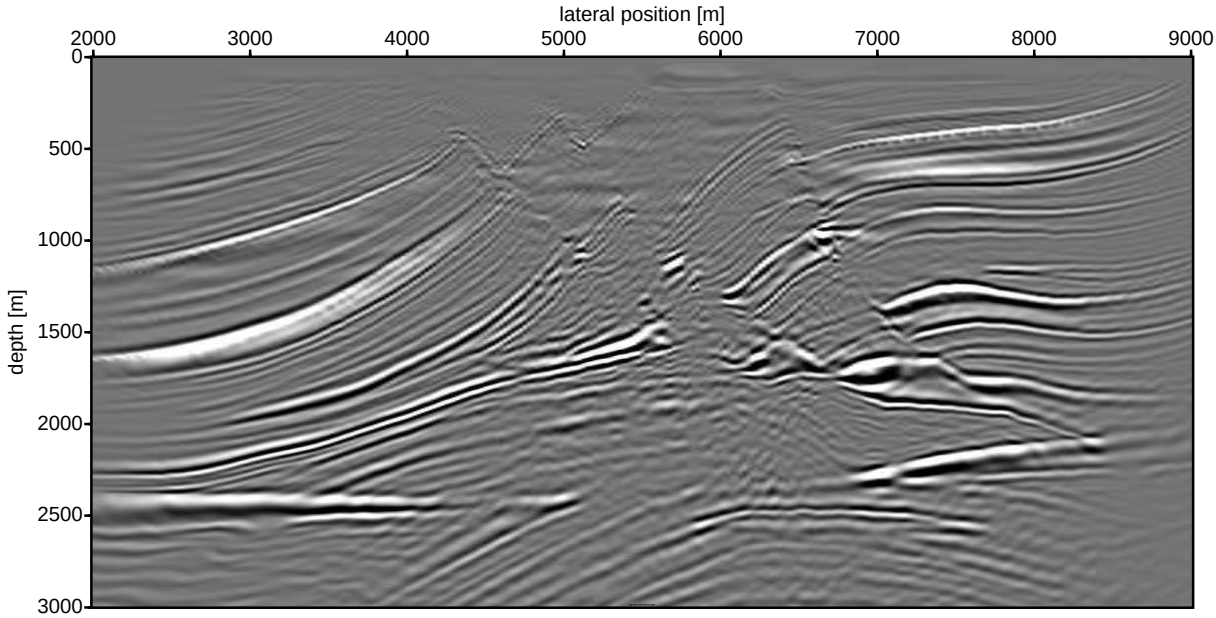
CPU's	Time (s)
1	31.6
2	16.9
4	9.9
8	6.4
16	4.3

Note that this code also includes a non-parallel part which takes about 2 seconds.

Marmousi model

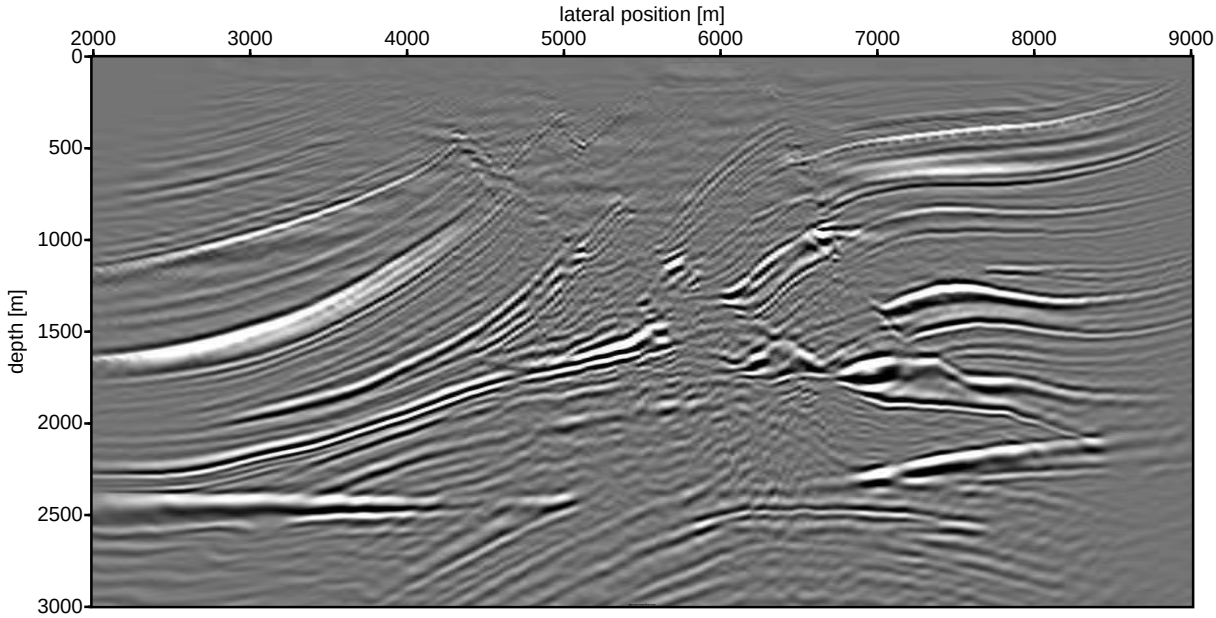


Marmousi model

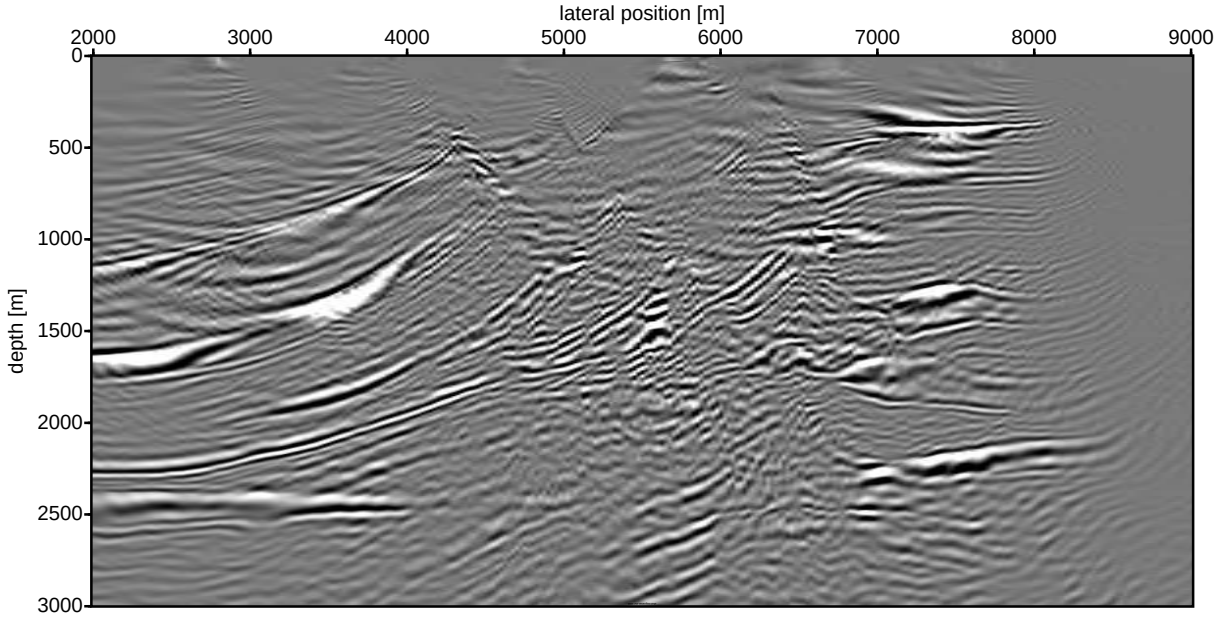


Shot record migration with $\Delta x_{src} = 25$ (all 240 shots)

shot record migration

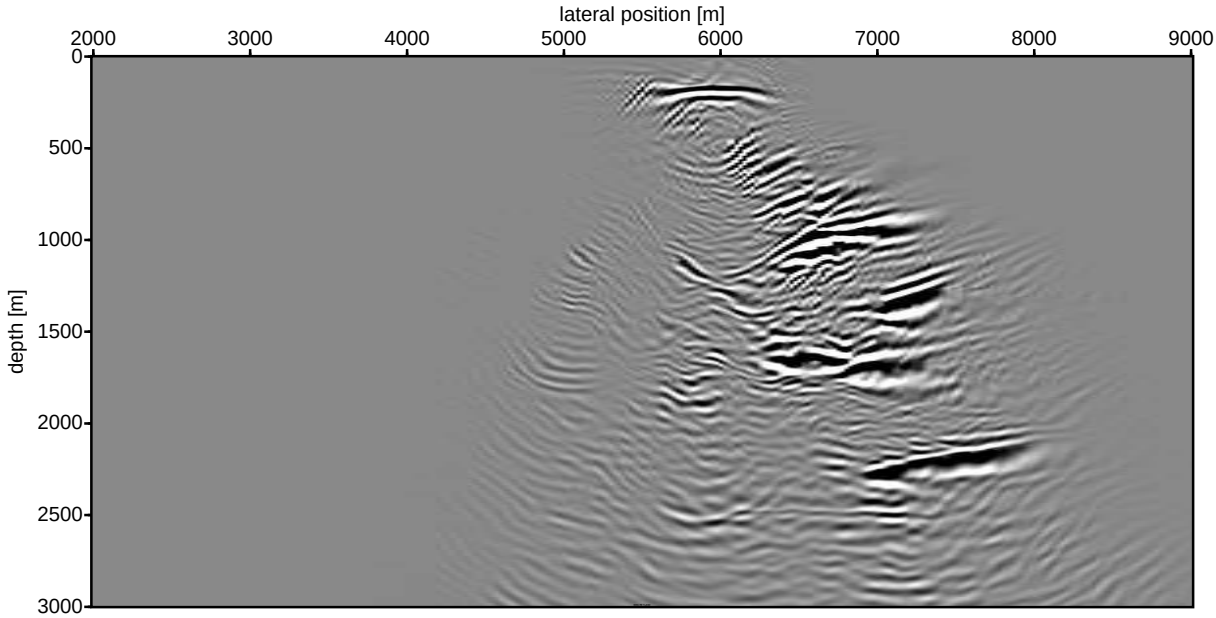


Shot record migration with $\Delta x_{src} = 100$ (61 shots)

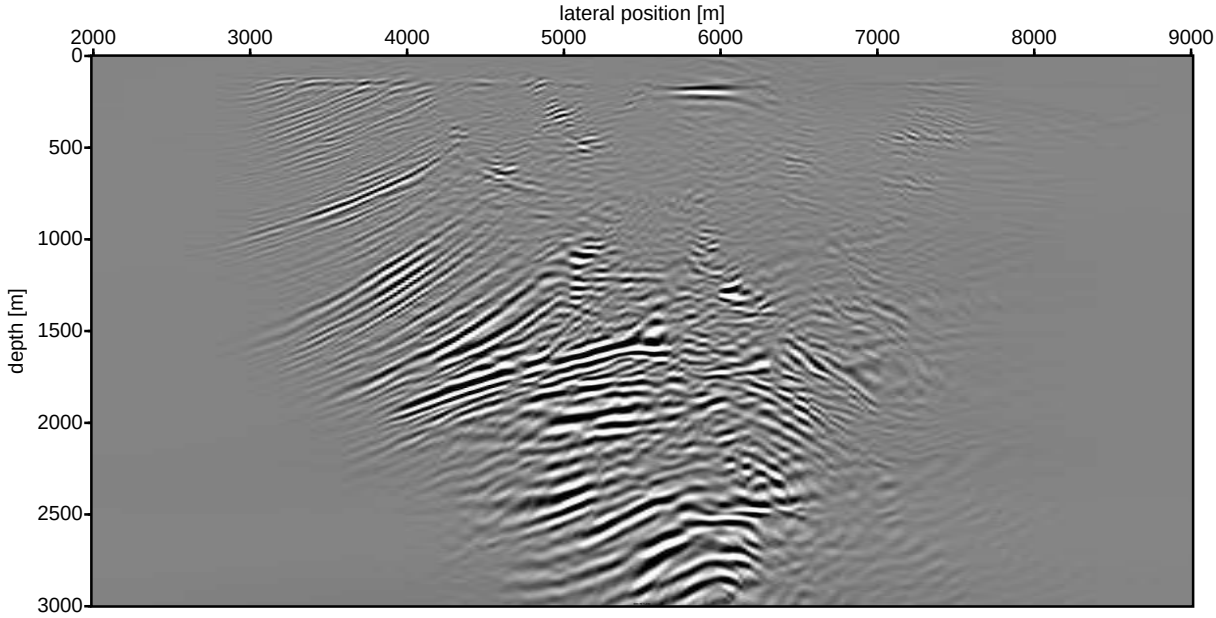


Shot record migration with $\Delta x_{src} = 1000$ (7 shots)

CFP gather migration

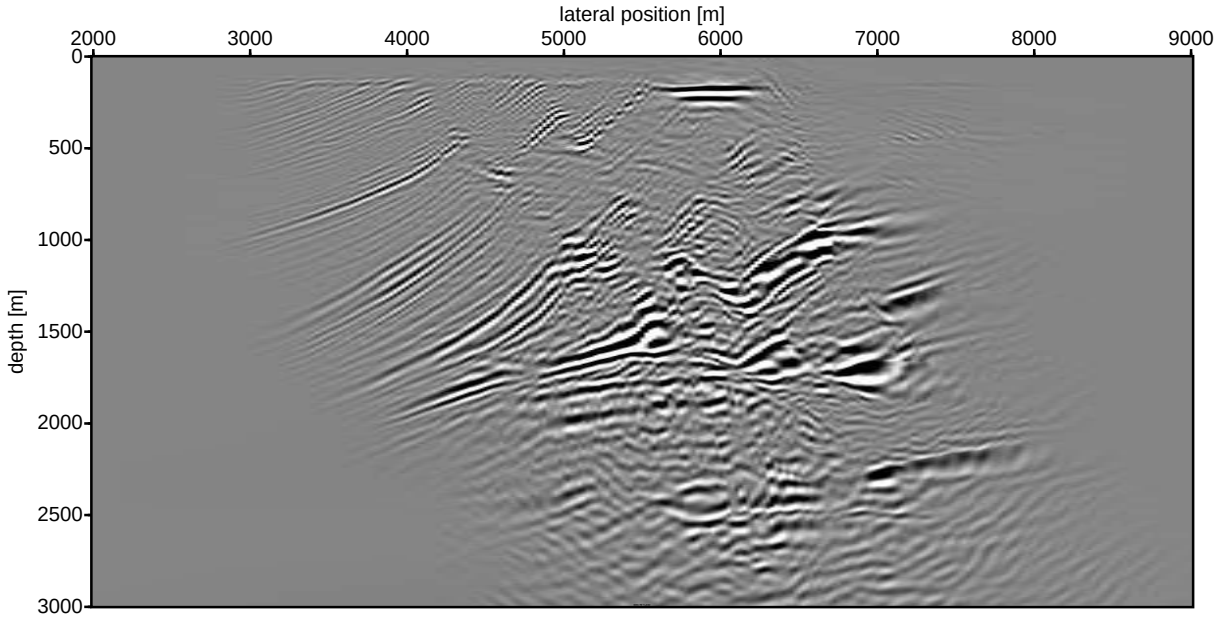


CFP gather migration with focus x=6000 and z=500

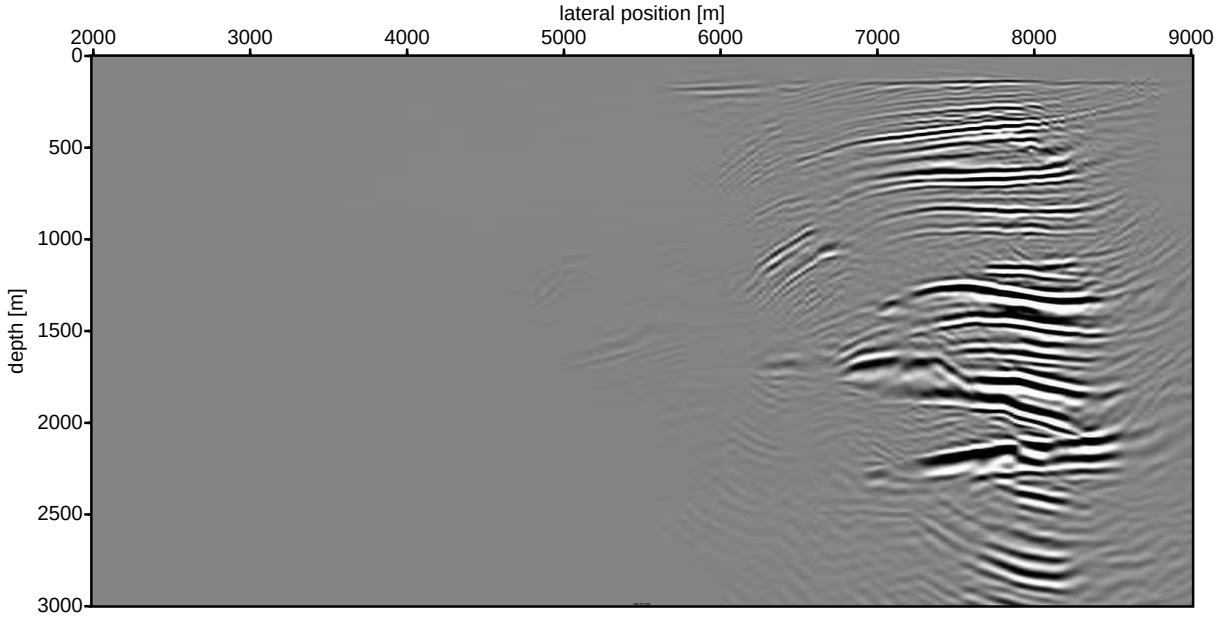


CFP gather migration with focus x=6000 and z=3000

CFP gather migration

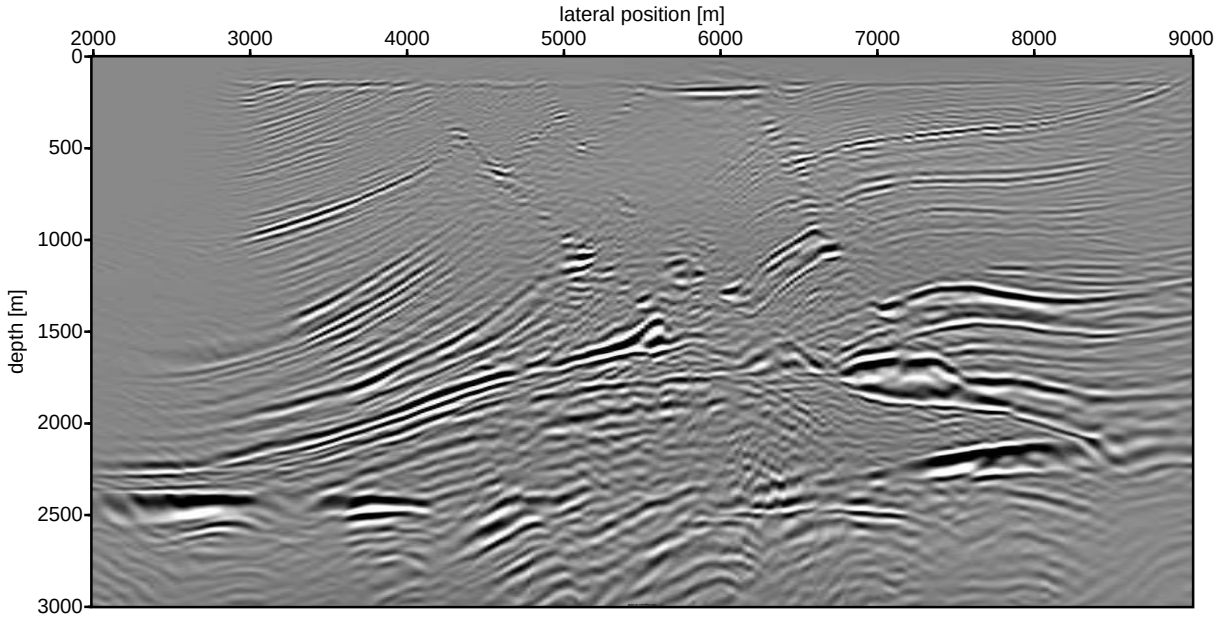


Focus at $x = 6000$ with $\Delta z = 500$ (6 gathers)

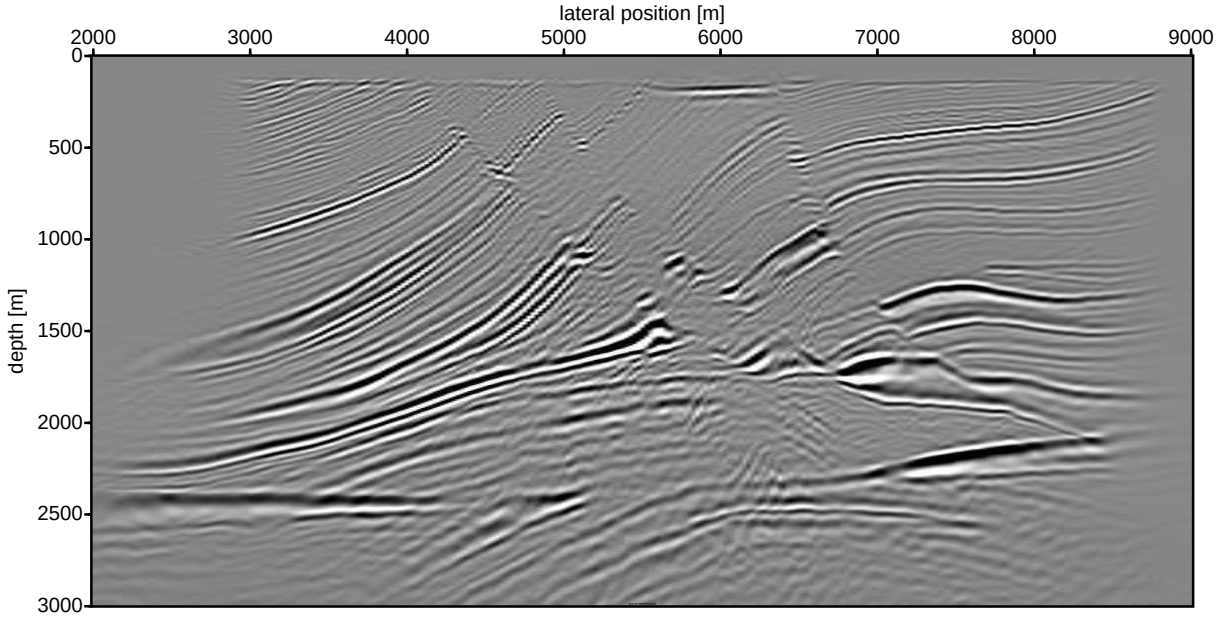


Focus at $x = 8000$ with $\Delta z = 500$ (6 gathers)

CFP gather migration

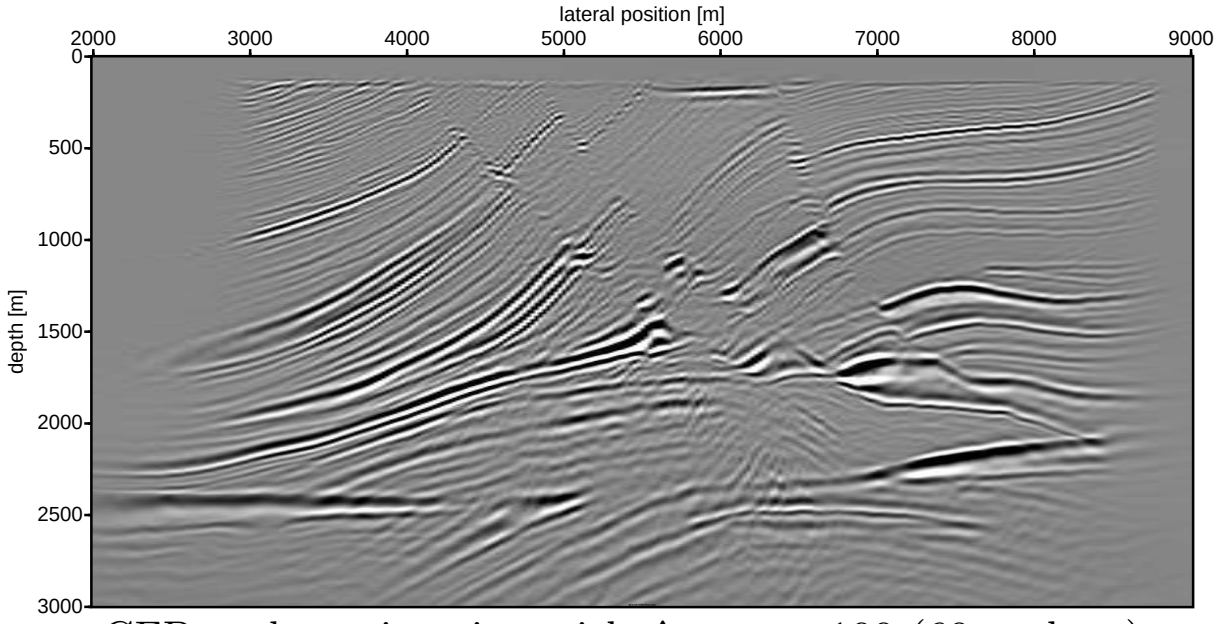


Focus at $z = 3000$ with $\Delta x_{cfp} = 1000$ (8 gathers)

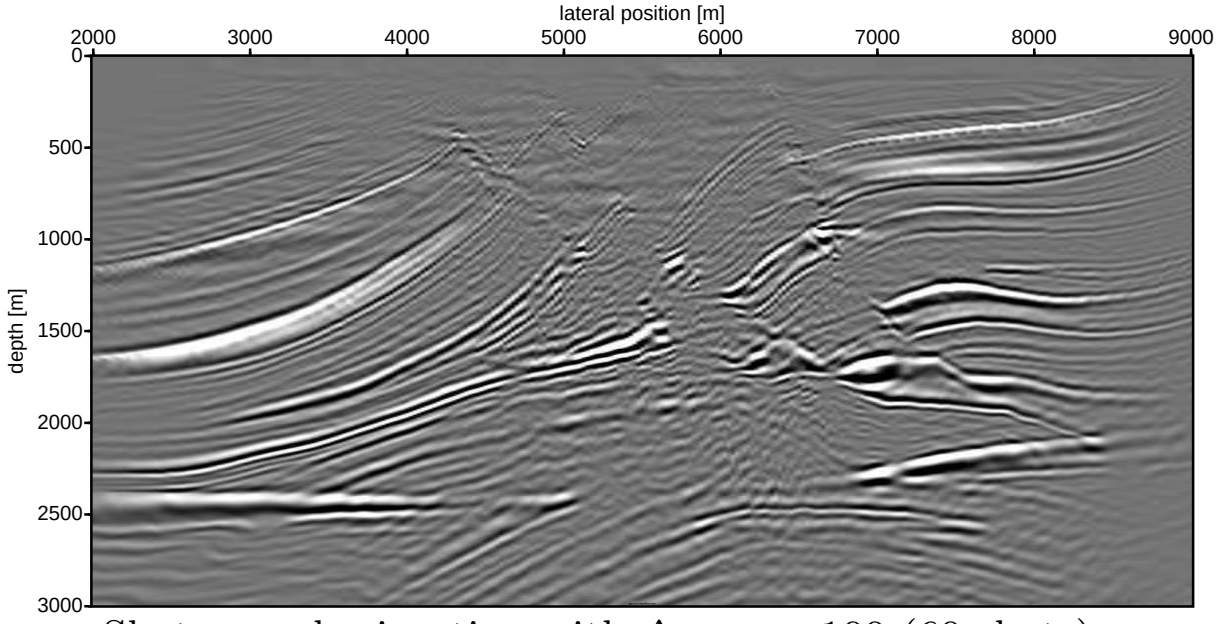


Focus at $z = 3000$ with $\Delta x_{cfp} = 250$ (29 gathers)

CFP vs shot migration

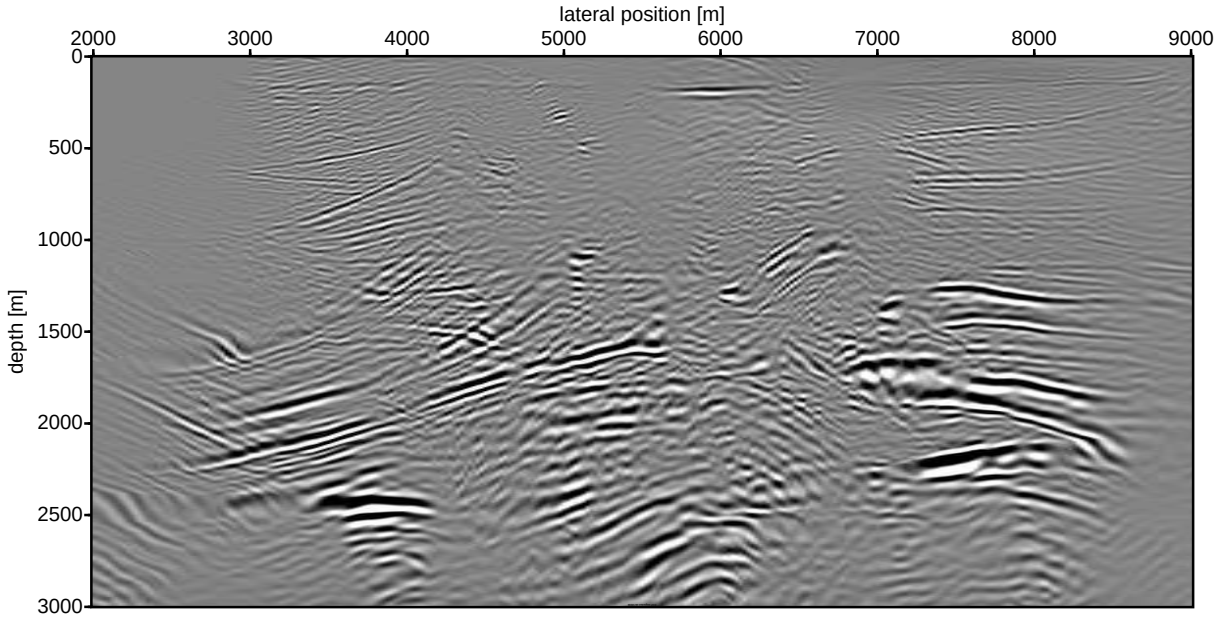


CFP gather migration with $\Delta x_{cfp} = 100$ (60 gathers)

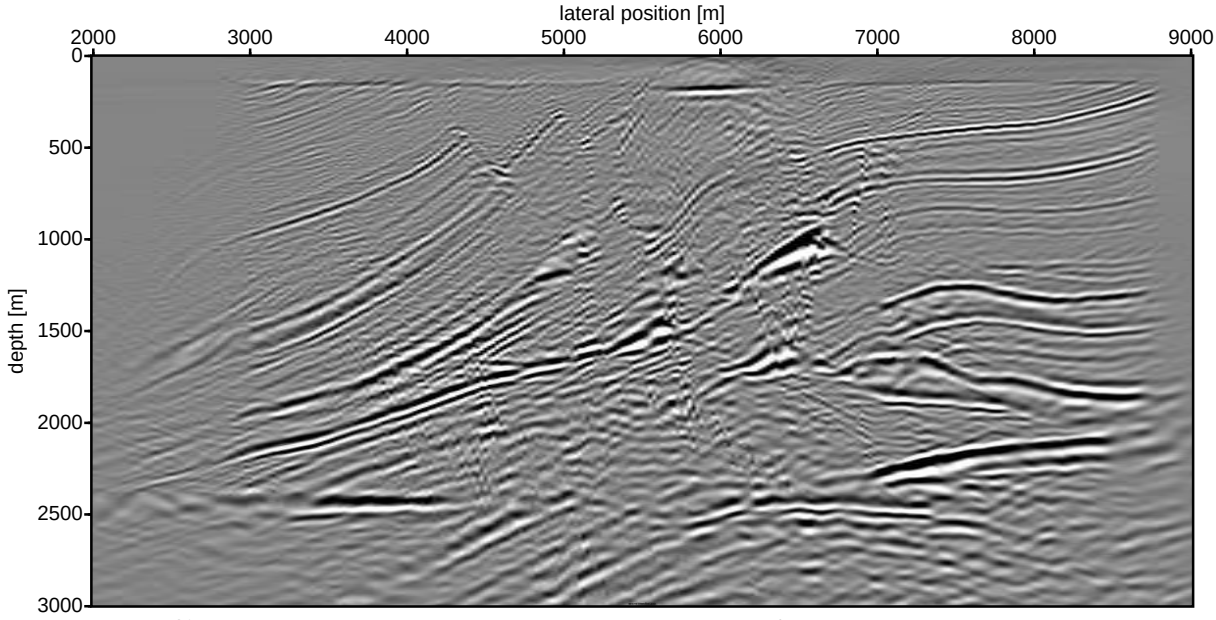


Shot record migration with $\Delta x_{src} = 100$ (60 shots)

CFP migration: addition

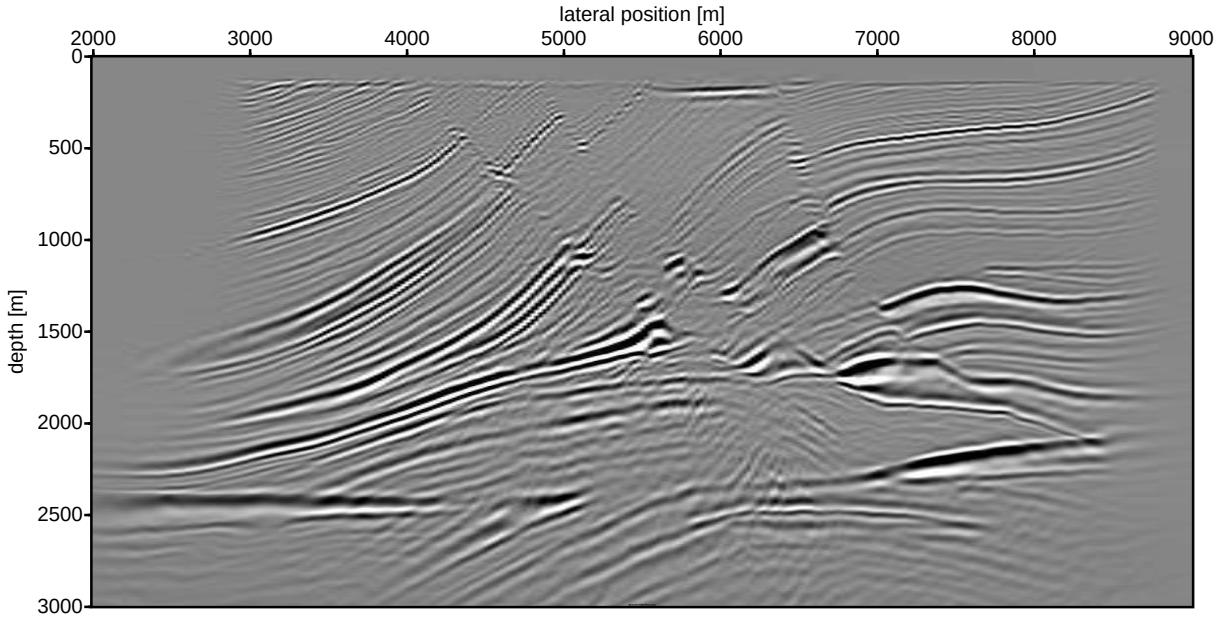


4 CFP gathers added together before migration.

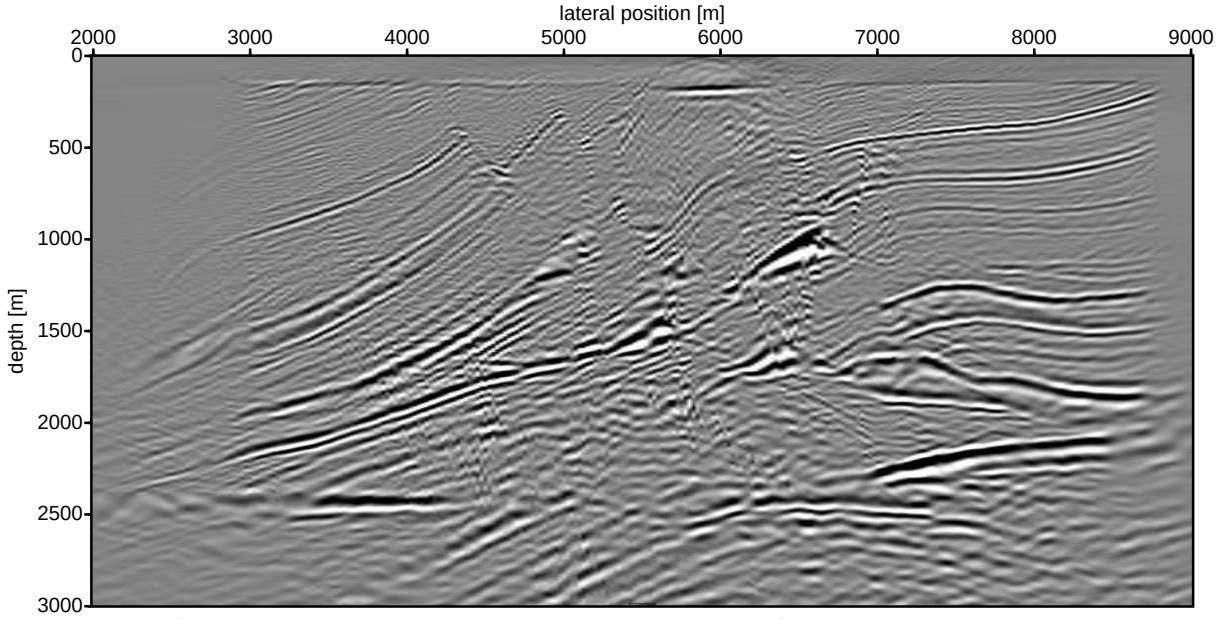


71 CFP gathers added together before migration.

Comparison CFP migrations



71 CFP gathers added together after migration



71 CFP gathers added together before migration.

Concluding remarks

- CFP gather migration is an efficient alternative for pre-stack shot record migration
- Other focal point distributions should be chosen to find an optimum distrubution which gives the highest image quality at the lowest computational cost (e.g a combination of shallow and deep focal points in a staggered way).
- Addition of CFP gathers before migration is not a good alternative if one considers the quality of the image.