Wavefield modelling and inversion in Matlab

Tristan van Leeuwen

SLIM
University of British Columbia

Algorithm design

Environment for developing FWI algorithms:

- Flexible
- Reasonably efficient and scalable
- Easy to maintain, test and debug

Algorithm design

- data cube and model `objects'

  `D, M`

- modelling operator

  `D = F(M)`

- Jacobian (linear operator)

  `D = J*M`

  `M = J'*D`

Algorithm design

Data/model objects:

- elementwise operations (`+,-,*`)
- reduction/transformation (`norm,fft`)
- may carry header information

Algorithm design

Linear operators

- forward/adjoint multiply (Ax, A^Tb)
- composition (`C = A*B`)
- coordinate free

Algorithm design

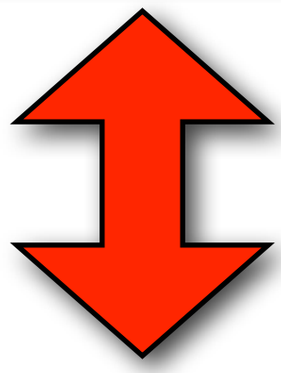
Solvers

- ask for operations on data
- ask for properties of data (e.g., norm)
- coordinate free

Algorithm design

1D linear operations can be combined via Kronecker products:

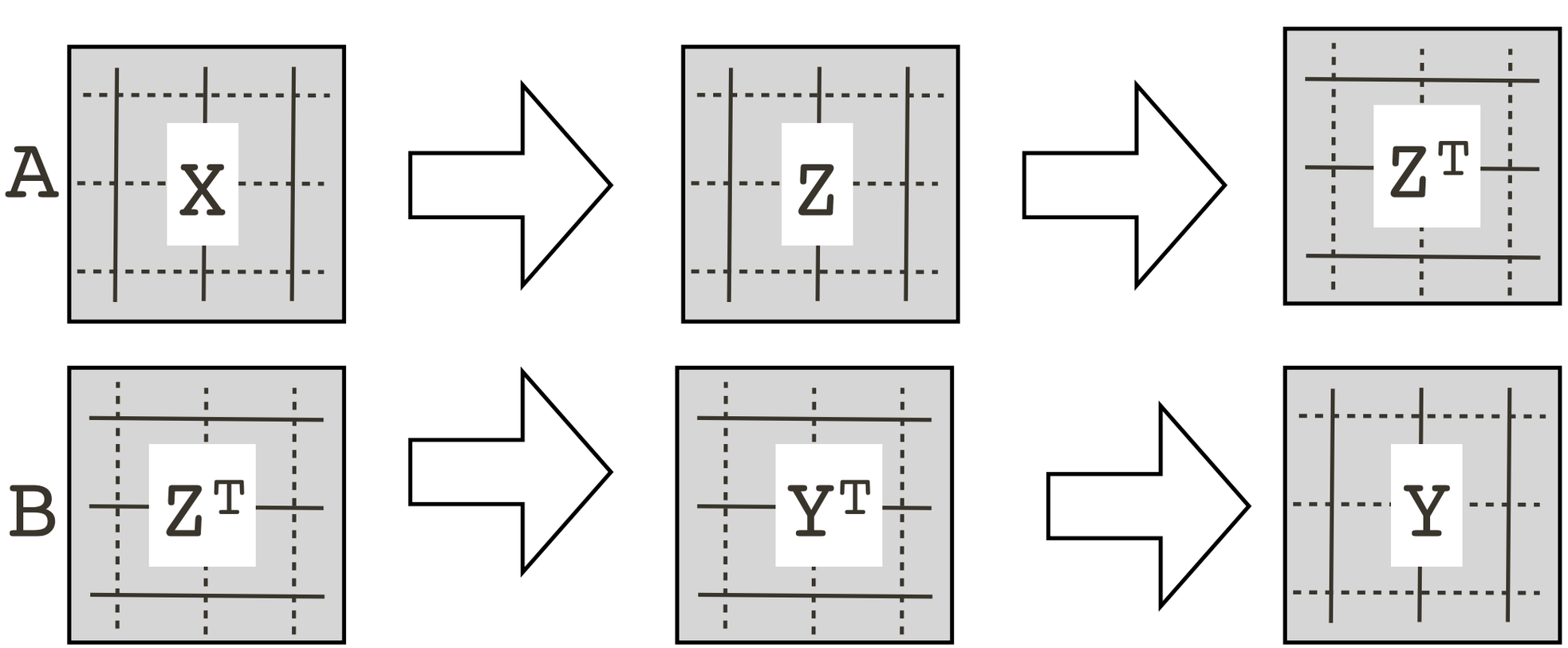
$$Y = AXB^T$$



$$\text{vec}(Y) = (B \otimes A)\text{vec}(X)$$

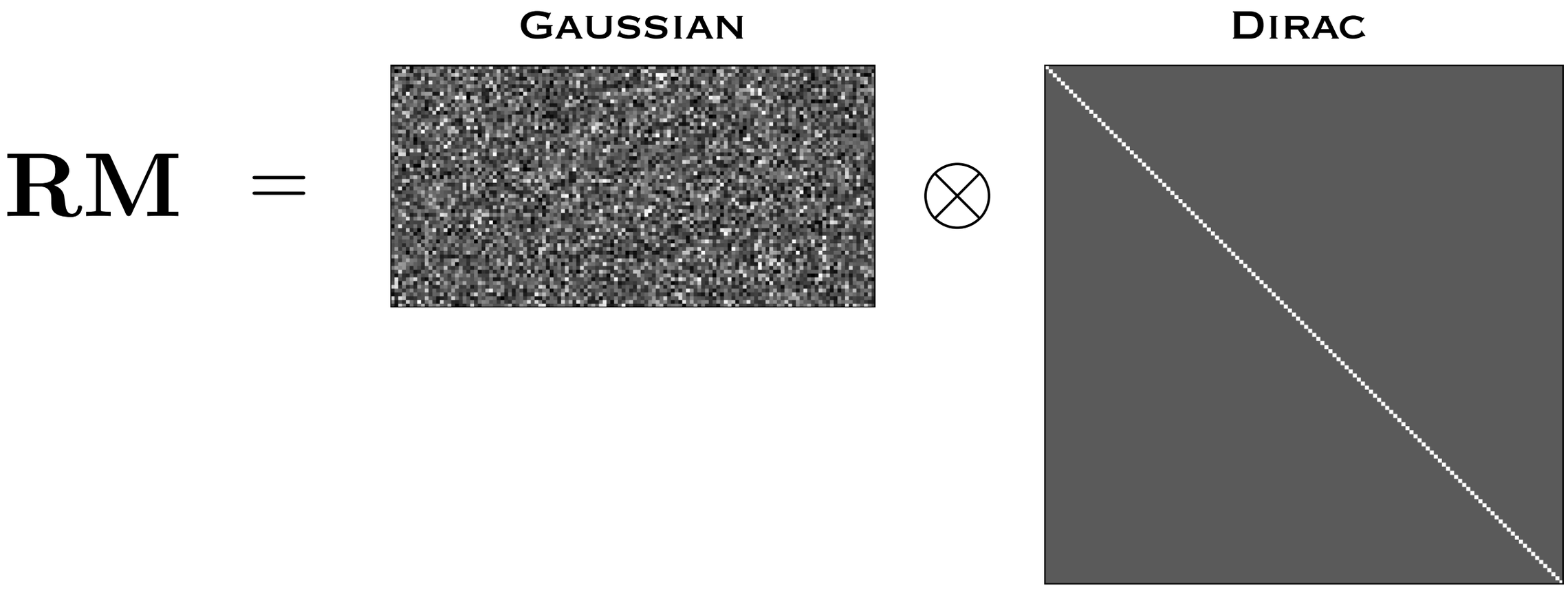
Algorithm design

Computations are easily parallelized over columns



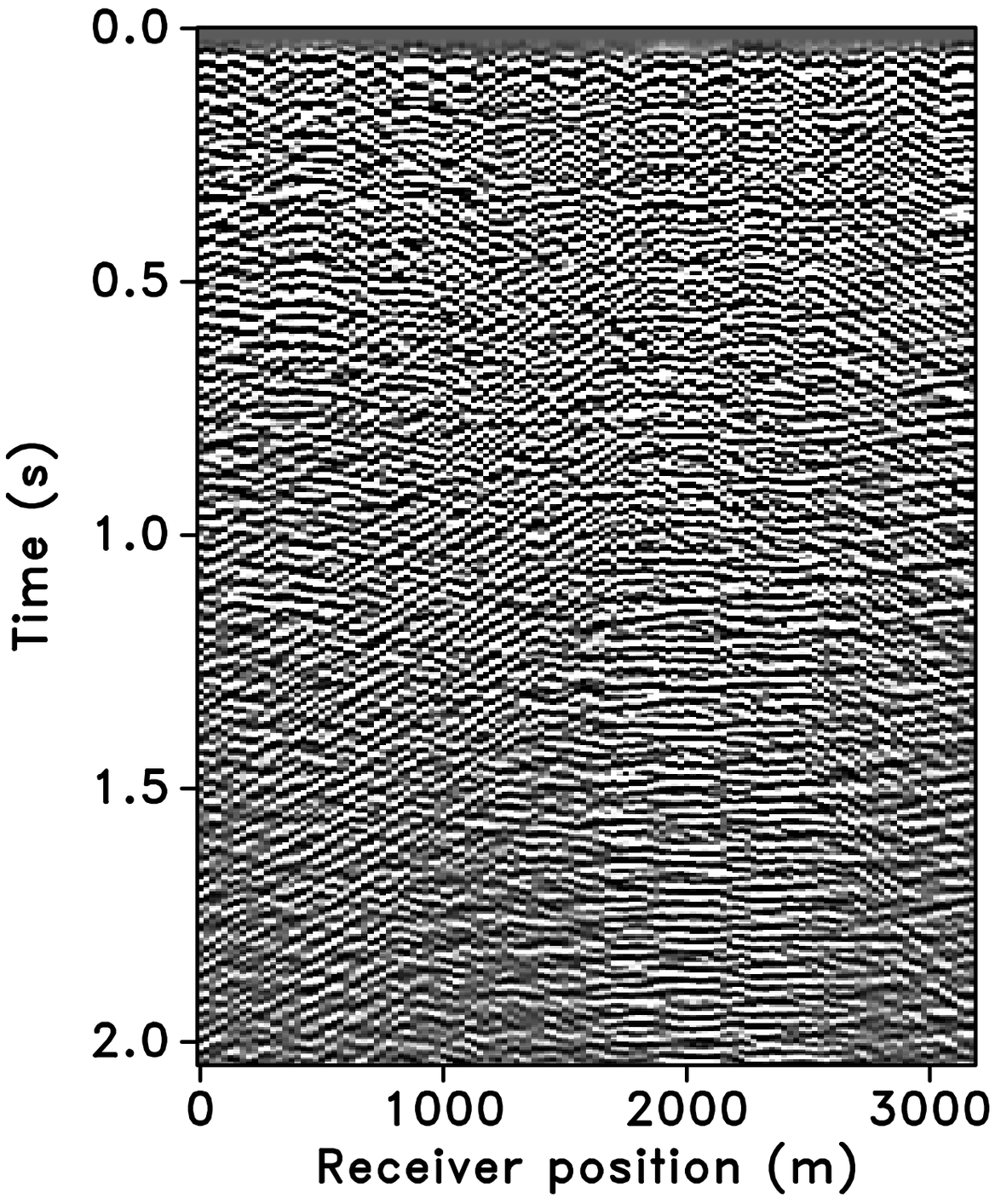
Sampling operator (RM)

```
% Subsampling ratio
p    = 1/2;

% Number of simultaneous-source experiments
nse  = round(p * dim(3));

% Dimensions for simultaneous-source experiments
sdim = [dim(1) dim(2) nse];

% Construct operator (RM)
RM   = opKron(opGaussian(sdim(3),dim(3)),opDirac(sdim(1)*sdim(2)));
```



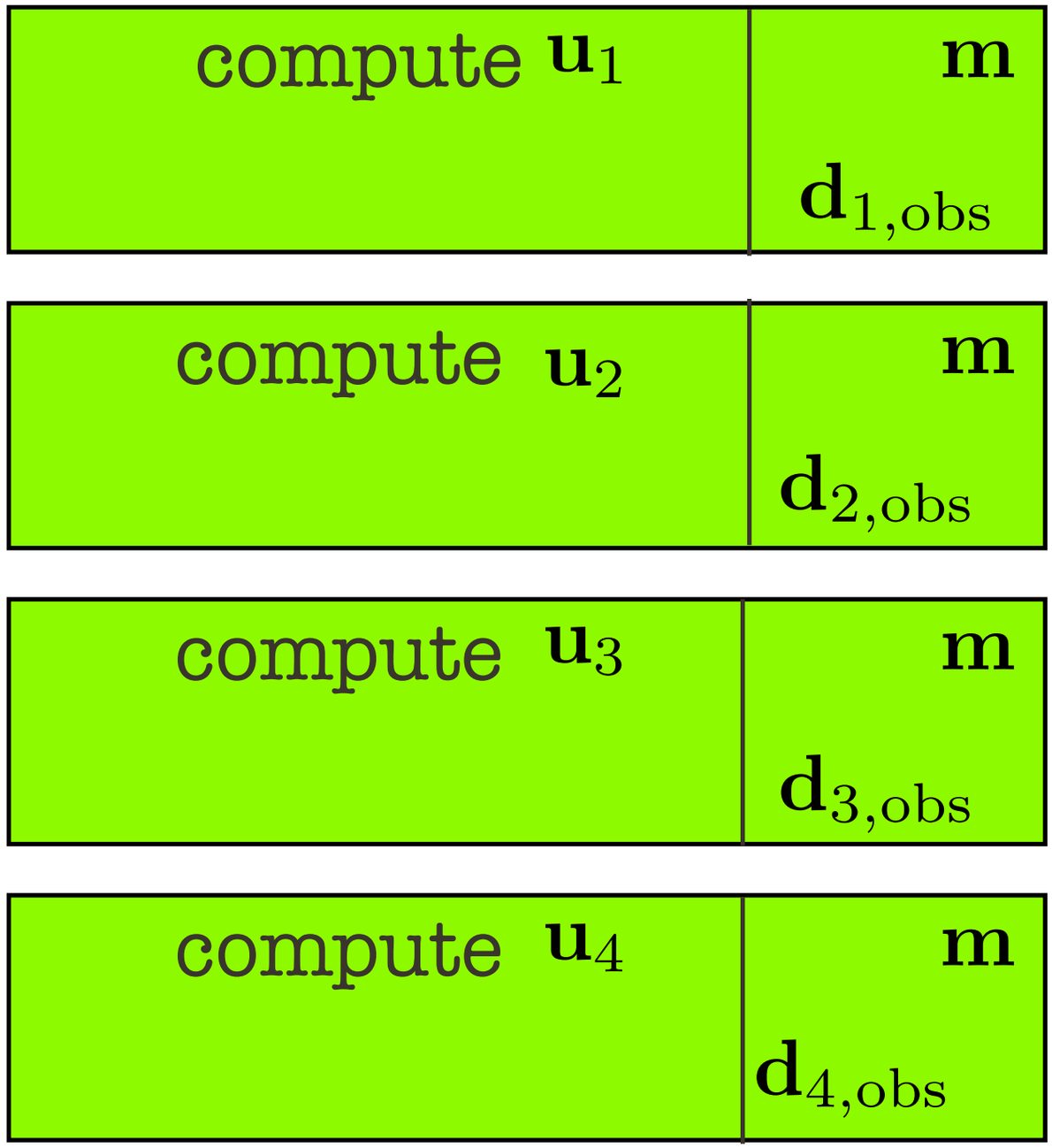
Simultaneous data

```
% Simultaneous measurements
b = RM*D(:);
```



Algorithm design

```
D=F(M)
R=D-Dobs
f=norm(R)
G=J*R
M=M-a*G
```



Algorithm design

```
D=F(M)
R=D-Dobs
f=norm(R)
G=J*R
M=M-a*G
```

$\mathbf{r}_1 = P\mathbf{u}_1 - \mathbf{d}_{1,\text{obs}}$	$\mathbf{u}_1\,\mathbf{m}$ $\mathbf{d}_{1,\text{obs}}$
$\mathbf{r}_2 = P\mathbf{u}_2 - \mathbf{d}_{2,\text{obs}}$	$\mathbf{u}_2\,\mathbf{m}$ $\mathbf{d}_{2,\text{obs}}$
$\mathbf{r}_3 = P\mathbf{u}_3 - \mathbf{d}_{3,\text{obs}}$	$\mathbf{u}_3\,\mathbf{m}$ $\mathbf{d}_{3,\text{obs}}$
$\mathbf{r}_4 = P\mathbf{u}_4 - \mathbf{d}_{4,\text{obs}}$	$\mathbf{u}_4\,\mathbf{m}$ $\mathbf{d}_{4,\text{obs}}$

Algorithm design

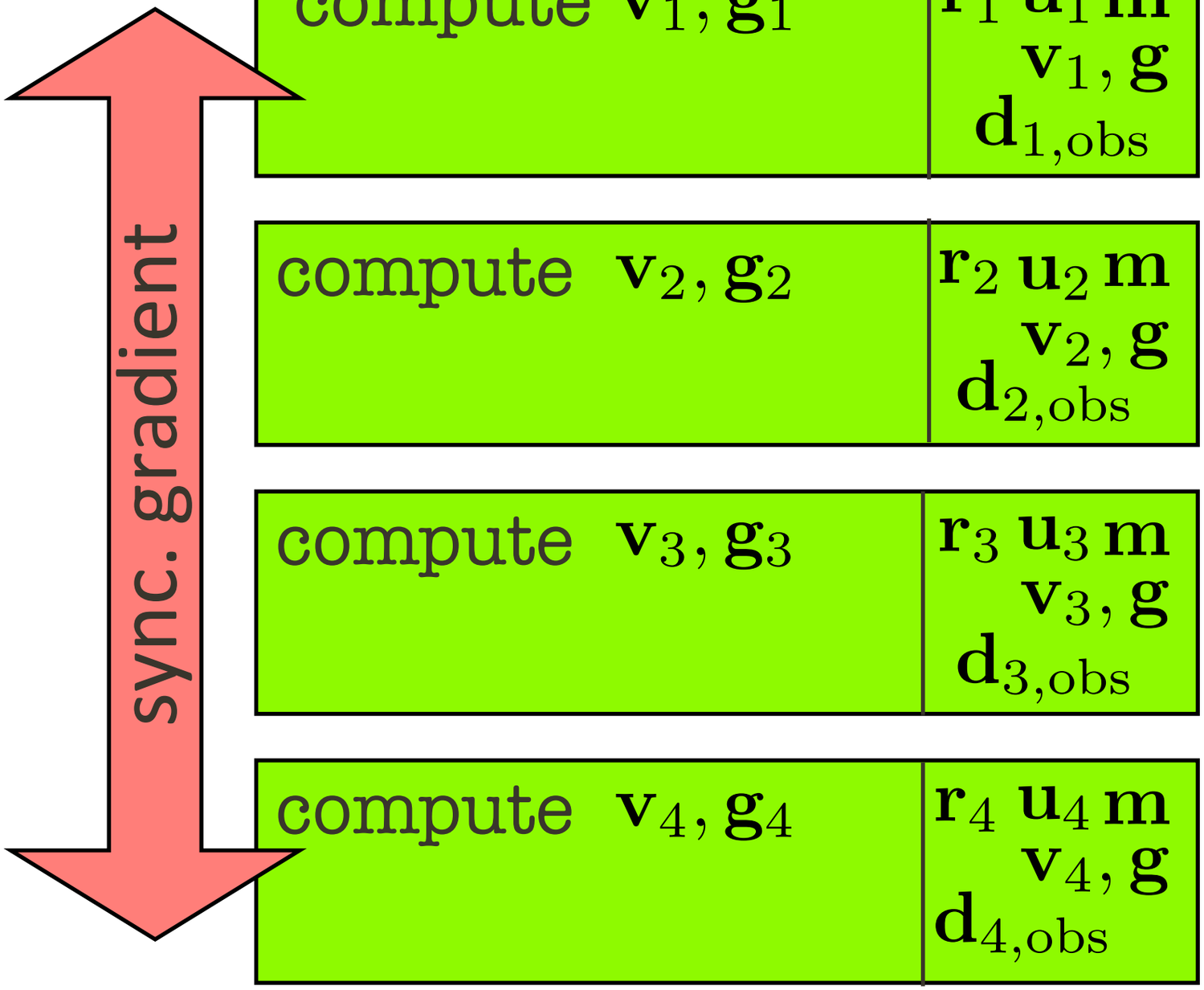
```
D=F(M)
R=D-Dobs
f=norm(R)
G=J*R
M=M-a*G
```

$$f = \sum_i f_i$$

$f_1 = \|\|\mathbf{r}_1\|\|_2^2$	$\mathbf{r}_1\ \mathbf{u}_1\ \mathbf{m}$ $\mathbf{d}_{1,\text{obs}}$
$f_2 = \|\|\mathbf{r}_2\|\|_2^2$	$\mathbf{r}_2\ \mathbf{u}_2\ \mathbf{m}$ $\mathbf{d}_{2,\text{obs}}$
$f_3 = \|\|\mathbf{r}_3\|\|_2^2$	$\mathbf{r}_3\ \mathbf{u}_3\ \mathbf{m}$ $\mathbf{d}_{3,\text{obs}}$
$f_4 = \|\|\mathbf{r}_4\|\|_2^2$	$\mathbf{r}_4\ \mathbf{u}_4\ \mathbf{m}$ $\mathbf{d}_{4,\text{obs}}$

Algorithm design

```
D=F(M)
R=D-Dobs
f=norm(R)
G=J*R
M=M-a*G
```



Algorithm design

```
D=F(M)
R=D-Dobs
f=norm(R)
G=J*R
M=M-a*G
```

a

$\mathbf{m} = \mathbf{m} - \alpha \mathbf{g}$	$\mathbf{r}_1\ \mathbf{u}_1\ \mathbf{m}$ $\mathbf{v}_1, \mathbf{g}$ $\mathbf{d}_{1,\mathrm{obs}}$

$\mathbf{m} = \mathbf{m} - \alpha \mathbf{g}$	$\mathbf{r}_2\ \mathbf{u}_2\ \mathbf{m}$ $\mathbf{v}_2, \mathbf{g}$ $\mathbf{d}_{2,\mathrm{obs}}$

$\mathbf{m} = \mathbf{m} - \alpha \mathbf{g}$	$\mathbf{r}_3\ \mathbf{u}_3\ \mathbf{m}$ $\mathbf{v}_3, \mathbf{g}$ $\mathbf{d}_{3,\mathrm{obs}}$

$\mathbf{m} = \mathbf{m} - \alpha \mathbf{g}$	$\mathbf{r}_4\ \mathbf{u}_4\ \mathbf{m}$ $\mathbf{v}_4, \mathbf{g}$ $\mathbf{d}_{4,\mathrm{obs}}$

Parallel Matlab

- Supports *distributed* arrays
- Most basic linear algebra is *overloaded* (parallel matvec's)
- No need to code MPI
- Low-level MPI commands are available if needed

Matlab implementation

modelling:

```
[D,J] = F(m,Q,model)
```

`D` - data cube as Matlab distributed array

`J` - Jacobian as SPOT operator

`m` - model as Matlab array

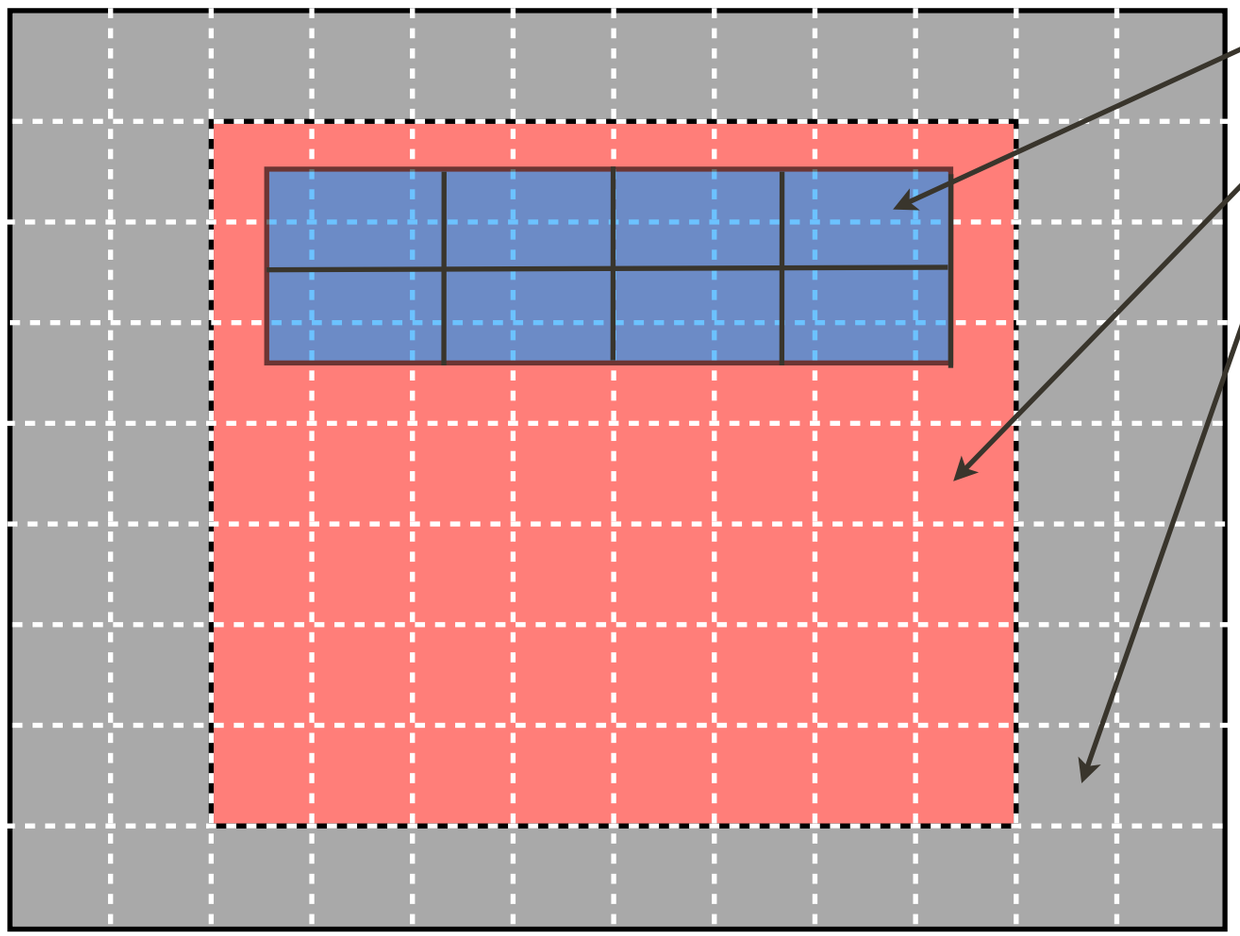
`Q` - source functions as matrix

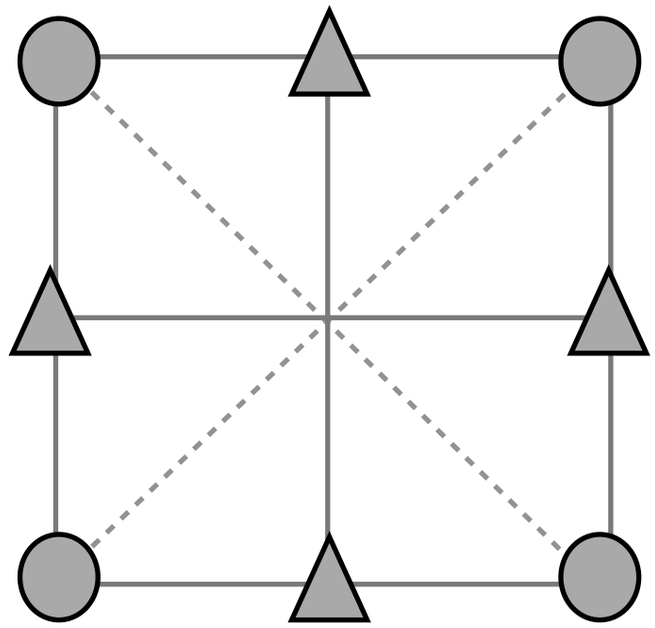
`model` - struct containing acquisition setup etc.

Matlab implementation

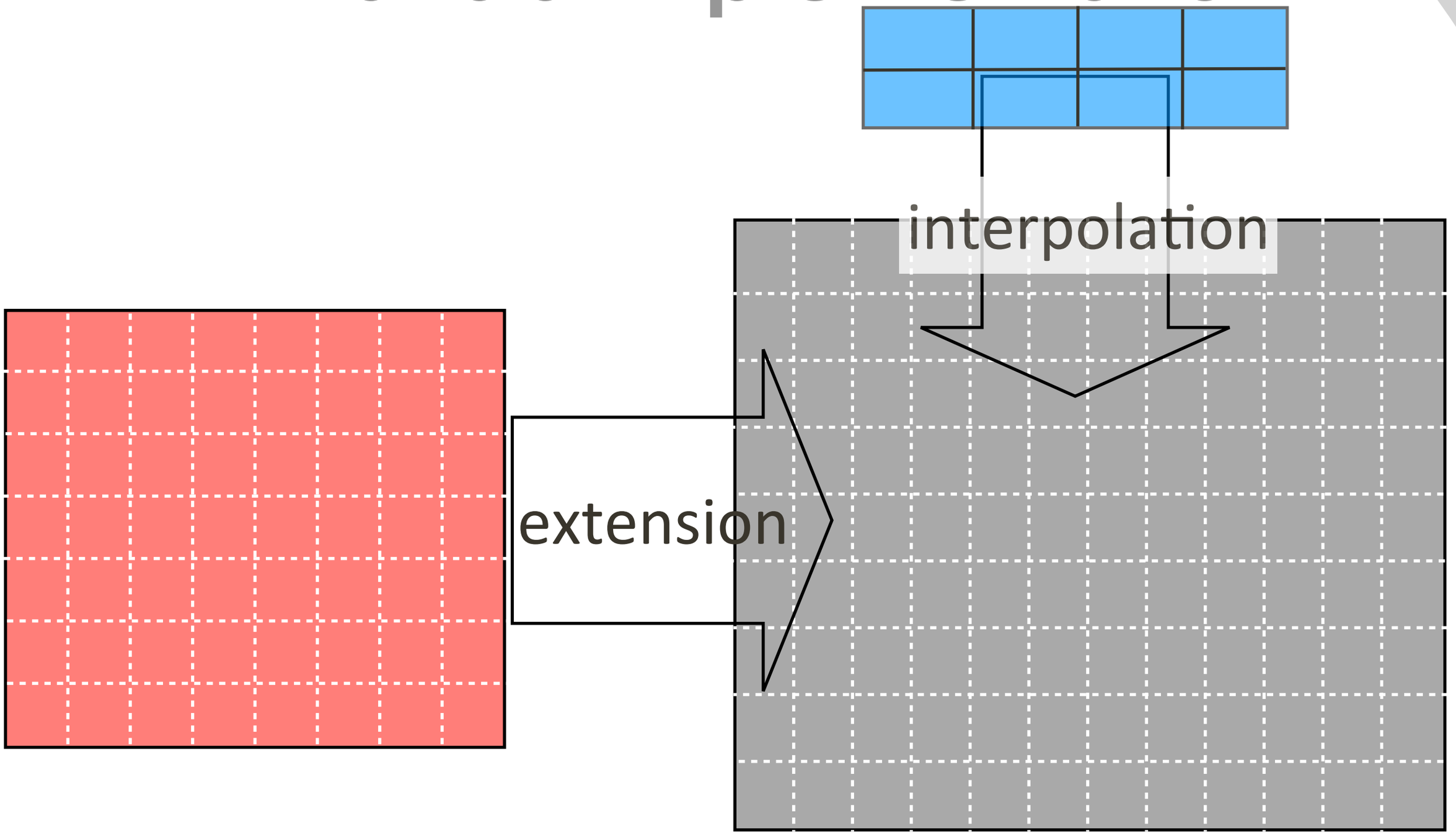
- frequency-domain FD modelling
- 9-point mixed-grid stencil
- `sponge' boundary
- source-injection and sampling via cubic interpolation
- parallel over frequencies
- Direct solver (Matlab->UMFPACK)

Matlab implementation





Matlab implementation



Matlab implementation

Born modelling and RTM

```
D    = J*dm;% calls J.multiply(dm,1)
dmt = J'*D;% calls J.multiply(D,-1)
```

remember, J is not a Matlab matrix but just an object.

Matlab implementation

- Jacobian based on linearization of the discretized system
- careful implementation of the adjoint
- recomputes wavefields

Matlab implementation

Testing

```
%test Jacobian
[D0,J0] = F(m0,Q,model);
D1      = F(m0+dm,Q,model);
dD      = J0*dm;
norm(D1 - D0 - dD)

%dottest: (Ax)'*b = x'*(A'*b)
(J*dm)'*dD
dm'(J'*dD)
```

Matlab implementation

least-squares migration

```
[D0,J0] = F(m0,Q,model);

%LSQR
m1 = LSQR(J0,D-D0);

%SPGL1
C  = opCurvelet(...);
x  = spgl1(J0*C',D-D0);
m2 = C'*x;
```

Matlab implementation

FWI

```
[f,g] = Jls(m,Q,Dobs,model)
%least-squares misfit

[D,J] = F(m,Q,model);
f     = .5*norm(D-Dobs).^2;
g     = J'*(D-Dobs);
```

Matlab implementation

FWI with source estimation

```
[f,g] = JlsSrc(m,Q,Dobs,model)
%least-squares misfit with source estimation

[D,J] = F(m,Q,model);
W     = SrcEst(D,Dobs);
f     = .5*norm(W*D-Dobs).^2;
g     = J'*(W'*df);
```

Matlab implementation

FWI with Students t

```
[f,g] = JStudentsT(m,Q,Dobs,model)
%Students T misfit

[D,J]   = F(m,Q,model);
[f,df] = StudentsT(D-Dobs);
g       = J'*df;
```

Matlab implementation

FWI with model parametrization

```
[f,g] = Jls(m,Q,Dobs,model)
%least-squares misfit

B      = opSpline(...);
[D,J] = F(B*m,Q,model);
f      = .5*norm(D-Dobs).^2;
g      = B'*J'*(D-Dobs);
```

Matlab implementation

Now, we can use *any* optimization code that uses only function values and gradients.

```
% function handle
fh = @(x)Jls(x,Q,D,model);

% optimization
mn = lbfgs(fh,m0);
```

Matlab implementation

```
m = GN(fh,m0,D)
% Gauss-Newton

m = m0;
for k = 1:maxit
  [Dk,Jk] = fh(m);
  dm      = LSQR(J,Dk - D);
  m       = m + dm;
end
```

Matlab implementation

Now, we can use *any* optimization code that uses only function values and gradients.

```
% function handle
fh = @(x)F(x,Q,model);

% optimization
mn = GN(fh,m0,D);
```

Future developments

The computational complexity of most of these algorithms lies in modelling, which we can do externally.

The memory complexity lies in the data volumes, which we do ***not*** have to import into Matlab

Future developments

- Interface to external modelling codes
- DataContainer object to access files as matlab arrays
- Interface to JavaSeis

Datacontainer

Matlab object that carries data and head

Command Window

```
>> help iCon.disp
 disp Display function for all dataContainers.
    disp(X) displays the header of our dataContainer and not the data
    itself. Displaying a dataContainer with default header values for double
    real serial array looks like:

    iCon dataContainer

    Variable Name:     unknown
    Variable Units:    unknown
    Dims:              n
    Size:              [ S1 S2 ... Sn]
    Origin:            [ O1 O2 ... On]
    Delta:             [ D1 D2 ... Dn]
    Label:             [ L1 L2 ... Ln]
    Unit:              [ U1 U2 ... Un]
    Precision:         double
    Complex:           No
    Distributed:       No

 Help for iCon/disp is inherited from superclass dataContainer

fx >>
```

SLIM

Datacontainer

Load data and define sampling intervals

Command Window

```
>> D = ReadSuFast('GulfOfSuez178.su');
>> whos D
  Name          Size                     Bytes  Class     Attributes

  D          1024x31684               259555328  double
```

Command Window

```
>> % Number of data samples
nt = 1024;          % Time
nr = 178;           % Receiver
ns = 178;           % Source


% Sampling intervals
dt = 0.004;         % Time
dr = 25;            % Receiver
ds = 25;            % Source
fx >>
```

Datacontainer

Construct data container

Command Window

```
>> D = iCon(reshape(D,nt,nr,ns), 'varName', 'Amplitude', 'varUnits', 'number','delta', [dt dr ds], ...
            'label', {'Time', 'Receiver position', 'Shot position'}, 'unit', {'s', 'm', 'm'});
>>
>> whos D
  Name         Size                     Bytes  Class    Attributes

  D         1024x178x178             259558978  iCon

>> D

D =

iCon dataContainer

Variable Name:     Amplitude
Variable Units:    number
Dims:              3
Size:              [ 1024 178 178 ]
Origin:            [ 0 0 0 ]
Delta:             [ 4.000000e-03 25 25 ]
Label:             [ Time Receiver position Shot position ]
Unit:              [ s m m ]
Precision:         double
Complex:           No
Distributed:       No
fx >>
```

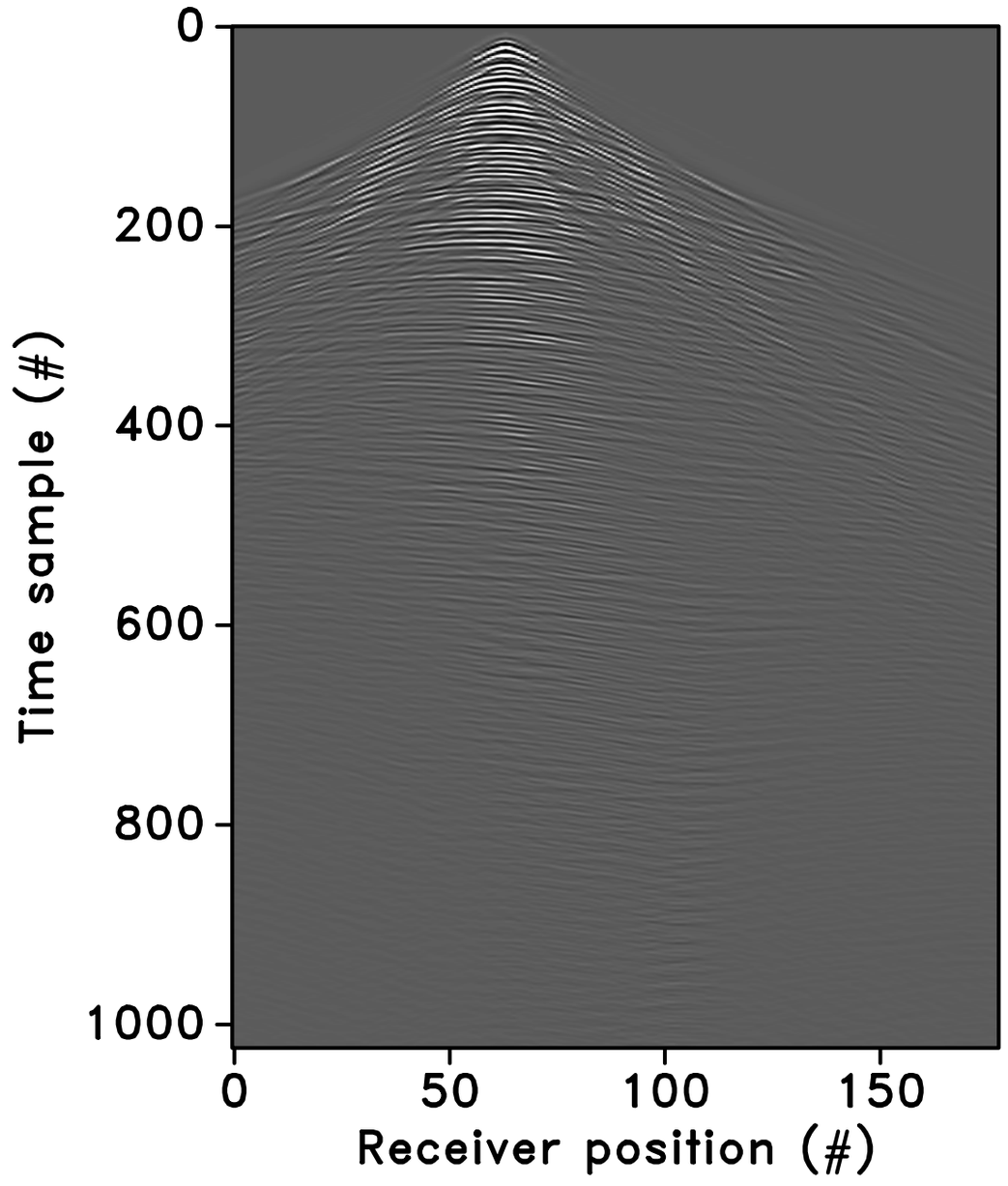
Datacontainer

Permute data

```
Command Window
>> D = permute(D,[3,2,1]);
>> whos D
  Name        Size                      Bytes  Class    Attributes

  D         178x178x1024             259558978  iCon

>> D

D =

iCon dataContainer

Variable Name:     Amplitude
Variable Units:    number
Dims:              3
Size:              [ 178 178 1024 ]
Origin:            [ 0 0 0 ]
Delta:             [ 25 25 4.000000e-03 ]
Label:             [ Shot position Receiver position Time ]
Unit:              [ m m s ]
Precision:         double
Complex:           No
Distributed:       No
fx >>
```

Datacontainer

```
Command Window
>> imagesc(D(:,:,64));
>>
```



Conclusions

- Use Matlab as `scripting language'
- allows us to quickly prototype and benefit from algorithms developed by `experts'
- codebase is modular and easy to maintain

Conclusions

- Use Matlab as `scripting language'
- allows us to quickly prototype and benefit from algorithms developed by `experts'
- codebase is modular and easy to maintain

Conclusions

- Overloading allows us to call external modelling code, and access data from external sources (disk, memory)
- no need to explicitly import data into Matlab
- Switch easily between matlab-arrays/data-containers and different modelling kernels

Coming soon...

Software release

Acknowledgements

Haneet Wason, for making some of the slides

This work was in part financially supported by the Natural Sciences and Engineering Research Council of Canada Discovery Grant (22R81254) and the Collaborative Research and Development Grant DNOISE II (375142-08). This research was carried out as part of the SINBAD II project with support from the following organizations: BG Group, BP, Chevron, ConocoPhillips, Petrobras, Total SA, BGP, PGS and WesternGeco.