Plasticity

R. Hielscher

Faculty of Mathematics,
Chemnitz University of Technology, Germany

MTEX Workshop 2019

What is a Tensor?

Tensors are used to described linear interactions between physical properties.

rank zero tensor scalar property, e.g. temperature

rank one tensor directional depended property, e.g. wave velocity

rank two tensors relationship between two vector fields, e.g. stress, strain, conductivity

rank three tensor relationship between a one and a two rank tensor, e.g. piezoelectricity

rank four tensor relationship between two two rank tensor, e.g. elasticity,

A tensor T of rank $s + t$ maps a tensor A of rank s onto a tensor B of rank t by the formula

$$B_{k_1,\ldots,k_t} = T_{k_1,k_2,\ldots,k_t,j_1,\ldots,j_s} A_{j_1,\ldots,j_s}$$

What is a Tensor?

Tensors are used to described linear interactions between physical properties.

rank zero tensor scalar property, e.g. temperature

rank one tensor directional depended property, e.g. wave velocity

rank two tensors relationship between two vector fields, e.g. stress, strain, conductivity

rank three tensor relationship between a one and a two rank tensor, e.g. piezoelectricity

rank four tensor relationship between two two rank tensor, e.g. elasticity,

A tensor T of rank $s + t$ maps a tensor A of rank s onto a tensor B of rank t by the formula

$$B_{k_1,\ldots,k_t} = T_{k_1,k_2,\ldots,k_t,j_1,\ldots,j_s} A_{j_1,\ldots,j_s}$$

What is a Tensor?

Tensors are used to described linear interactions between physical properties.

rank zero tensor scalar property, e.g. temperature

rank one tensor directional depended property, e.g. wave velocity

rank two tensors relationship between two vector fields, e.g. stress, strain, conductivity

rank three tensor relationship between a one and a two rank tensor, e.g. piezoelectricity

rank four tensor relationship between two two rank tensor, e.g. elasticity,

A tensor T of rank $s + t$ maps a tensor A of rank s onto a tensor B of rank t by the formula

$$B_{k_1,\ldots,k_t} = T_{k_1,k_2,\ldots,k_t,j_1,\ldots,j_s} A_{j_1,\ldots,j_s}$$

A Simple Example

```
sigma = stressTensor.uniaxial(vector3d.Z)
```

```
sigma = stressTensor (show methods, plot)
  unit: MPa
  rank: 2 (3 x 3)

  0 0 0
  0 0 0
  0 0 1
```

A Simple Example

```
sigma = stressTensor.uniaxial(vector3d.Z)
```

```
M = [[1.45  0.00  0.19];...
     [0.00  2.11  0.00];...
     [0.19  0.00  1.79]];
sigma = stressTensor(M, 'unit', 'MPa');
```

```
sigma = stressTensor (show methods, plot)
  unit: MPa
  rank: 2 (3 x 3)

  1.45     0  0.19
     0  2.11     0
  0.19     0  1.79
```

A Simple Example

```
sigma = stressTensor.uniaxial(vector3d.Z)
```

```
M = [[1.45  0.00  0.19];...
     [0.00  2.11  0.00];...
     [0.19  0.00  1.79]];
sigma = stressTensor(M, 'unit', 'MPa');
```

```
n = vector3d.X % normal direction
```

```
n = vector3d (show methods, plot)
  size: 1 x 1
  x y z
  1 0 0
```

A Simple Example

```
sigma = stressTensor.uniaxial(vector3d.Z)
```

```
M = [[1.45  0.00  0.19];...
     [0.00  2.11  0.00];...
     [0.19  0.00  1.79]];
sigma = stressTensor(M, 'unit', 'MPa');
```

```
n = vector3d.X % normal direction
```

the stress vector $T^{\vec{n}}$ of plane $\vec{n} = \{1, 0, 0\}$, is computed by $T_j^{\vec{n}} = \sigma_{ij}\vec{n}_i$.

A Simple Example

```
sigma = stressTensor.uniaxial(vector3d.Z)
```

```
M = [[1.45  0.00  0.19];...
     [0.00  2.11  0.00];...
     [0.19  0.00  1.79]];
sigma = stressTensor(M, 'unit', 'MPa');
```

```
n = vector3d.X % normal direction
```

the stress vector $T^{\vec{n}}$ of plane $\vec{n} = \{1, 0, 0\}$, is computed by $T_j^{\vec{n}} = \sigma_{ij}\vec{n}_i$.

```
T = EinsteinSum(sigma,[-1  1],n,-1)
```

```
T = tensor (show methods, plot)
  unit: MPa
  rank: 1 (3)

 1.45
    0
 0.19
```

A Simple Example

```
sigma = stressTensor.uniaxial(vector3d.Z)
```

```
M = [[1.45  0.00  0.19];...
     [0.00  2.11  0.00];...
     [0.19  0.00  1.79]];
sigma = stressTensor(M, 'unit', 'MPa');
```

```
n = vector3d.X % normal direction
```

the stress vector $T^{\vec{n}}$ of plane $\vec{n} = \{1, 0, 0\}$, is computed by $T_j^{\vec{n}} = \sigma_{ij}\vec{n}_i$.

```
T = EinsteinSum(sigma,[-1  1],n,-1)
```

```
T = sigma' * n
```

```
ans = vector3d (show methods, plot)
 size: 1 x 1
     x     y     z
  1.45     0  0.19
```

Einstein Summation

The scalar magnitudes of the normal stress σ_N and the shear stress σ_S are given as

$$\sigma_N = T_i^{\vec{n}} \vec{n}_i = \sigma_{ij} \vec{n}_i \vec{n}_j \quad \text{and} \quad \sigma_S = \sqrt{T_i^{\vec{n}} T_i^{\vec{n}} - \sigma_N^2}.$$

```
sigmaN = EinsteinSum(T,-1,n,-1)
sigmaS = sqrt(EinsteinSum(T,-1,T,-1) - sigmaN^2)
```

```
sigmaN =

    1.4500

sigmaS =

    0.1900
```

Einstein Summation

The scalar magnitudes of the normal stress σ_N and the shear stress σ_S are given as

$$\sigma_N = T_i^{\vec{n}} \vec{n}_i = \sigma_{ij} \vec{n}_i \vec{n}_j \quad \text{and} \quad \sigma_S = \sqrt{T_i^{\vec{n}} T_i^{\vec{n}} - \sigma_N^2}.$$

```
sigmaN = EinsteinSum(T,-1,n,-1)
sigmaS = sqrt(EinsteinSum(T,-1,T,-1) - sigmaN^2)
```

```
sigmaN =

    1.4500

sigmaS =

    0.1900
```

Visualization

For a second order tensor σ_{ij} its directional magnitude $R(\vec{x})$ is

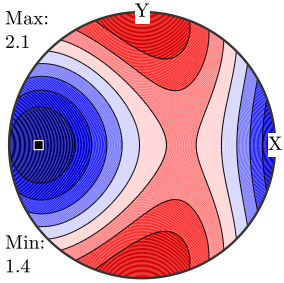
$$R(\vec{x}) = \sigma_{ij}\vec{x}_i\vec{x}_j.$$

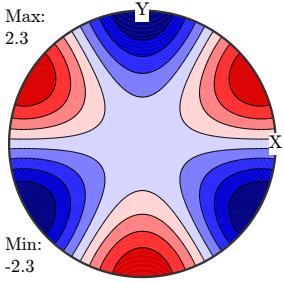
```
R = EinsteinSum(sigma,[-1 -2],x,-1,x,-2)
```

```
R = directionalMagnitude(sigma)
```

```
[value,pos] = min(R)
```

```
plot(sigma.directionalMagnitude)
annotate(pos)
plot(sigma,'minmax')
mtexColorMap blue2red
```





Visualization

For a second order tensor σ_{ij} its directional magnitude $R(\vec{x})$ is

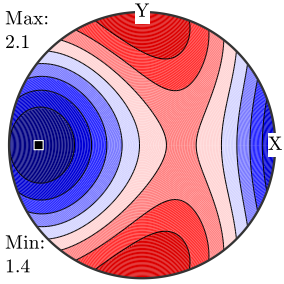
$$R(\vec{x}) = \sigma_{ij}\vec{x}_i\vec{x}_j.$$

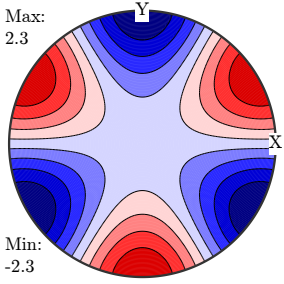
```
R = EinsteinSum(sigma,[-1 -2],x,-1,x,-2)
```

```
R = directionalMagnitude(sigma)
```

```
[value,pos] = min(R)
```

```
plot(sigma.directionalMagnitude)
annotate(pos)
plot(sigma,'minmax')
mtexColorMap blue2red
```





Visualization

For a second order tensor σ_{ij} its directional magnitude $R(\vec{x})$ is

$$R(\vec{x}) = \sigma_{ij}\vec{x}_i\vec{x}_j.$$

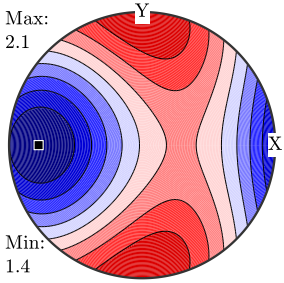
```
R = EinsteinSum(sigma,[-1 -2],x,-1,x,-2)
```

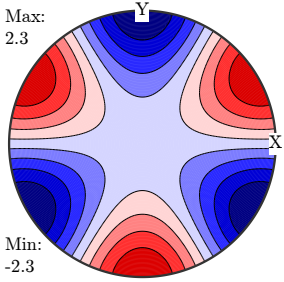
```
R = directionalMagnitude(sigma,x)
```

```
R = directionalMagnitude(sigma)

[value,pos] = min(R)

plot(sigma.directionalMagnitude)
annotate(pos)
plot(sigma,'minmax')
mtexColorMap blue2red
```





Visualization

For a second order tensor σ_{ij} its directional magnitude $R(\vec{x})$ is

$$R(\vec{x}) = \sigma_{ij}\vec{x}_i\vec{x}_j.$$

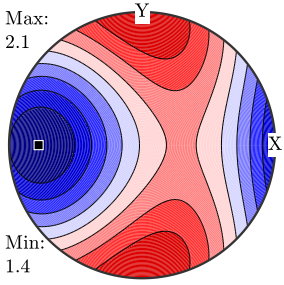
```
R = EinsteinSum(sigma,[-1 -2],x,-1,x,-2)
```

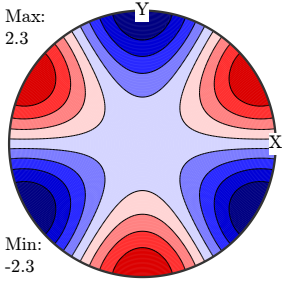
```
R = directionalMagnitude(sigma)
```

```
R = S2FunHarmonic (show methods, plot)
 bandwidth: 2
 antipodal: true
```

```
[value,pos] = min(R)
```

```
plot(sigma.directionalMagnitude)
annotate(pos)
plot(sigma,'minmax')
mtexColorMap blue2red
```





Visualization

For a second order tensor σ_{ij} its directional magnitude $R(\vec{x})$ is

$$R(\vec{x}) = \sigma_{ij}\vec{x}_i\vec{x}_j.$$

```
R = EinsteinSum(sigma,[-1 -2],x,-1,x,-2)
```

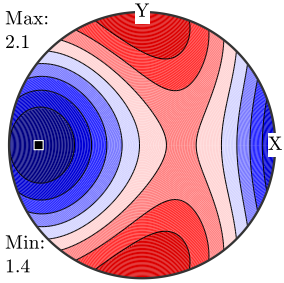
```
R = directionalMagnitude(sigma)
```

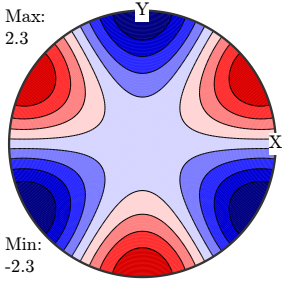
```
[value,pos] = min(R)
```

```
value =
    1.3652

pos = vector3d
 size: 1 x 1
          x          y          z
  -0.918733          0   0.394879
```

```
plot(sigma.directionalMagnitude)
annotate(pos)
```





Visualization

For a second order tensor σ_{ij} its directional magnitude $R(\vec{x})$ is

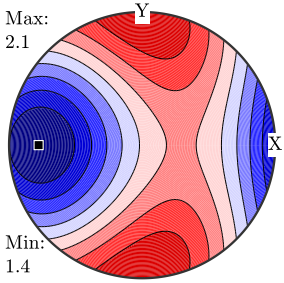
$$R(\vec{x}) = \sigma_{ij}\vec{x}_i\vec{x}_j.$$

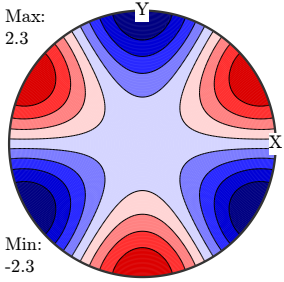
```
R = EinsteinSum(sigma,[-1 -2],x,-1,x,-2)
```

```
R = directionalMagnitude(sigma)
```

```
[value,pos] = min(R)
```

```
plot(sigma.directionalMagnitude)
annotate(pos)
plot(sigma,'minmax')
mtexColorMap blue2red
```





Field Tensors vs. Matter Tensors

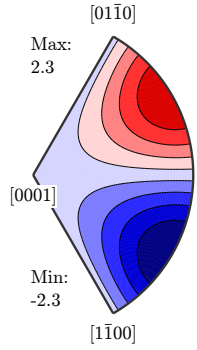
matter tensors:

- in crystal coordinates
- describe physical properties like: electrical or thermal conductivity, magnetic permeability, compliance

```
CS = crystalSymmetry.load('Quartz')
P = piezoElectricityTensor(M, CS, 'unit', 'C/N')
```

```
P = piezoElectricityTensor (show methods, plot)
  unit   : C/N
  rank   : 3 (3 x 3 x 3)
  mineral: Quartz (321, X||a*, Y||b, Z||c)

  tensor in compact matrix form:
     0     0     0 -0.67     0   4.6
   2.3  -2.3     0     0  0.67     0
     0     0     0     0     0     0
```



field tensors:

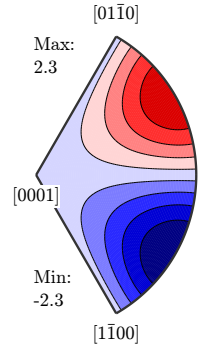
- in specimen coordinates

Field Tensors vs. Matter Tensors

matter tensors:

- in crystal coordinates
- describe physical properties like: electrical or thermal conductivity, magnetic permeability, compliance

```
CS = crystalSymmetry.load('Quartz')
P = piezoElectricityTensor(M, CS, 'unit', 'C/N')
```



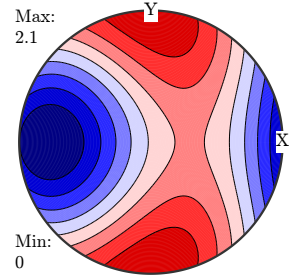
field tensors:

- in specimen coordinates
- describe applied forces like: stress, electric field

```
sigma = stressTensor.uniaxial(vector3d.Z)
```

```
sigma = stressTensor (show methods, plot)
  rank: 2 (3 x 3)

  0  0  0
  0  0  0
  0  0  1
```



Rotating Tensors

Consider the piezoelectricity tensor

```
P = piezoElectricityTensor(M, CS)
```

```
P = piezoElectricityTensor (show methods, plot)
  unit   : C/N
  rank   : 3 (3 x 3 x 3)
  mineral: Quartz (321, X||a*, Y||b, Z||c)

  tensor in compact matrix form:
     0     0     0 -0.67     0   4.6
   2.3  -2.3     0     0  0.67     0
     0     0     0     0     0     0
```

Remember orientations transforms crystal into specimen coordinates

```
ori = orientation.byEuler(10*degree,20*degree,0,CS)
ori * P
```

Contrary, an inverse orientation transforms specimen coordinates into crystal coordinates.

```
inv(ori) * sigma
```

Rotating Tensors

Consider the piezoelectricity tensor

```
P = piezoElectricityTensor(M, CS)
```

Remember orientations transforms crystal into specimen coordinates

```
ori = orientation.byEuler(10*degree,20*degree,0,CS)
ori * P
```

```
ans = piezoElectricityTensor (show methods, plot)
  unit: C/N
  rank: 3 (3 x 3 x 3)

  tensor in compact matrix form:
  -1.08   1.25  -0.17  -0.01   1.47   3.72
   1.92  -1.63  -0.29  -1.29   1.10   1.86
   0.76  -0.66  -0.09  -0.46   0.30   0.43
```

Contrary, an inverse orientation transforms specimen coordinates into crystal coordinates.

```
inv(ori) * sigma
```

Rotating Tensors

Consider the piezoelectricity tensor

```
P = piezoElectricityTensor(M, CS)
```

Remember orientations transforms crystal into specimen coordinates

```
ori = orientation.byEuler(10*degree,20*degree,0,CS)
ori * P
```

Contrary, an inverse orientation transforms specimen coordinates into crystal coordinates.

```
inv(ori) * sigma
```

```
ans = stressTensor (show methods, plot)
  unit    : MPa
  rank    : 2 (3 x 3)
  mineral: Quartz (321, X||a*, Y||b, Z||c)

  1.47  0.17  0.14
  0.17  2.03 -0.12
  0.14 -0.12  1.85
```

Average Tensors

The average tensorial property of a specimen is the mean of matter tensors rotated according to each grain orientation o_m, $m = 1, \dots, M$.

The Voigt and the Reuss averages of a tensor T are defines as

$$\langle T \rangle^{\text{Voigt}} = \sum_{m=1}^{M} o_m \cdot T, \quad \langle T \rangle^{\text{Reuss}} = \left[\sum_{m=1}^{M} o_m \cdot T^{-1} \right]^{-1}.$$

For EBSD data this is computed by

```
[TVoigt, TReus, THill] = calcTensor(ebsd,T)
```

and for an ODF by

```
[TVoigt, TReus, THill] = calcTensor(odf,T)
```

Average Tensors

The average tensorial property of a specimen is the mean of matter tensors rotated according to each grain orientation o_m, $m = 1, \dots, M$.

The Voigt and the Reuss averages of a tensor T are defines as

$$\langle T \rangle^{\text{Voigt}} = \sum_{m=1}^{M} o_m \cdot T, \quad \langle T \rangle^{\text{Reuss}} = \left[\sum_{m=1}^{M} o_m \cdot T^{-1} \right]^{-1}.$$

For EBSD data this is computed by

```
[TVoigt, TReus, THill] = calcTensor(ebsd,T)
```

and for an ODF by

```
[TVoigt, TReus, THill] = calcTensor(odf,T)
```

Average Tensors

The average tensorial property of a specimen is the mean of matter tensors rotated according to each grain orientation o_m, $m = 1, \dots, M$.

The Voigt and the Reuss averages of a tensor T are defines as

$$\langle T \rangle^{\text{Voigt}} = \sum_{m=1}^{M} o_m \cdot T, \quad \langle T \rangle^{\text{Reuss}} = \left[\sum_{m=1}^{M} o_m \cdot T^{-1} \right]^{-1}.$$

For EBSD data this is computed by

```
[TVoigt, TReus, THill] = calcTensor(ebsd,T)
```

and for an ODF by

```
[TVoigt, TReus, THill] = calcTensor(odf,T)
```

Average Tensors

The average tensorial property of a specimen is the mean of matter tensors rotated according to each grain orientation o_m, $m = 1, \dots, M$.

The Voigt and the Reuss averages of a tensor T are defines as

$$\langle T \rangle^{\text{Voigt}} = \sum_{m=1}^{M} o_m \cdot T, \quad \langle T \rangle^{\text{Reuss}} = \left[\sum_{m=1}^{M} o_m \cdot T^{-1} \right]^{-1}.$$

For EBSD data this is computed by

```
[TVoigt, TReus, THill] = calcTensor(ebsd,T)
```

and for an ODF by

```
[TVoigt, TReus, THill] = calcTensor(odf,T)
```

The Velocity Gradient Tensor

describes the local rate of deformation

```
L = velocityGradientTensor.uniaxial(vector3d.Z)
```

```
L = velocityGradientTensor (show methods, plot)
  rank: 2 (3 x 3)

 -0.5    0    0
    0 -0.5    0
    0    0    1
```

```
d_exp = vector3d.Y; d_compr = vector3d.X;
L = velocityGradientTensor.pureShear(d_exp,d_compr)
```

```
d = vector3d.Y; n = vector3d.X;
L = velocityGradientTensor.simpleShear(d,n)
```

```
rot = rotation.byAxisAngle(vector3d.Z,10*degree)
L = velocityGradientTensor.spin(rot)
```

The Velocity Gradient Tensor

describes the local rate of deformation

```
L = velocityGradientTensor.uniaxial(vector3d.Z)
```

```
d_exp = vector3d.Y; d_compr = vector3d.X;
L = velocityGradientTensor.pureShear(d_exp,d_compr)
```

```
L = velocityGradientTensor (show methods, plot)
  rank: 2 (3 x 3)

  -2   0   0
   0   2   0
   0   0   0
```

```
d = vector3d.Y; n = vector3d.X;
L = velocityGradientTensor.simpleShear(d,n)
```

```
rot = rotation.byAxisAngle(vector3d.Z,10*degree)
L = velocityGradientTensor.spin(rot)
```

The Velocity Gradient Tensor

describes the local rate of deformation

```
L = velocityGradientTensor.uniaxial(vector3d.Z)
```

```
d_exp = vector3d.Y; d_compr = vector3d.X;
L = velocityGradientTensor.pureShear(d_exp,d_compr)
```

```
d = vector3d.Y; n = vector3d.X;
L = velocityGradientTensor.simpleShear(d,n)
```

```
L = velocityGradientTensor (show methods, plot)
  rank: 2 (3 x 3)

 0 0 0
 2 0 0
 0 0 0
```

```
rot = rotation.byAxisAngle(vector3d.Z,10*degree)
L = velocityGradientTensor.spin(rot)
```

The Velocity Gradient Tensor

describes the local rate of deformation

```
L = velocityGradientTensor.uniaxial(vector3d.Z)
```

```
d_exp = vector3d.Y; d_compr = vector3d.X;
L = velocityGradientTensor.pureShear(d_exp,d_compr)
```

```
d = vector3d.Y; n = vector3d.X;
L = velocityGradientTensor.simpleShear(d,n)
```

```
rot = rotation.byAxisAngle(vector3d.Z,10*degree)
L = velocityGradientTensor.spin(rot)
```

```
L = spinTensor (show methods, plot)
  rank: 2 (3 x 3)

 *10^-2
       0 -17.453        0
  17.453        0        0
       0        0        0
```

Decomposition of the Velocity Gradient Tensor

The deformation gradient tensor can be decomposed into a sum of a symmetric and an antisymmetric tensor

```
W = L.antiSym
D = L.sym
```

```
W = spinTensor (show methods, plot)
  rank: 2 (3 x 3)

   0  -1  0
   1   0  0
   0   0  0

D = strainRateTensor (show methods, plot)
  rank: 2 (3 x 3)

  0  1  0
  1  0  0
  0  0  0
```

Decomposition of the Velocity Gradient Tensor

The deformation gradient tensor can be decomposed into a sum of a symmetric and an antisymmetric tensor

```
W = L.antiSym
D = L.sym
```

the antisymmetric portion models the spin

```
rotation(W)
```

```
ans = rotation (show methods, plot)
  size: 1 x 1

  Bunge Euler angles in degree
  phi1    Phi     phi2     Inv.
  57.3      0        0        0
```

The Deformation Gradient Tensor $\mathbf{F}$

describes the local deformation in continuum mechanics and satisfies

$$\dot{\mathbf{F}}(t) = \mathbf{L}(t)\,\mathbf{F}(t)$$

In case of constant deformation $\mathbf{L}(t) = \mathbf{L}$ we obtain

$$\mathbf{F}(t) = \exp(t \cdot \mathbf{L})$$

```
F = expm(L * t)
```

In the case of time dependent deformation we have

$$\mathbf{F}(t) = \exp\left(\int_0^t \mathbf{L}(\tau)\,\mathrm{d}\tau\right)$$

```
tau = linspace(0,t); d_tau = tau(2)-tau(1);
L_tau = (t-tau) .* L
F = expm(cumsum(L_tau * d_tau))
```

The Deformation Gradient Tensor **F**

describes the local deformation in continuum mechanics and satisfies

$$\dot{\mathbf{F}}(t) = \mathbf{L}(t)\,\mathbf{F}(t)$$

In case of constant deformation $\mathbf{L}(t) = \mathbf{L}$ we obtain

$$\mathbf{F}(t) = \exp(t \cdot \mathbf{L})$$

```
F = expm(L * t)
```

In the case of time dependent deformation we have

$$\mathbf{F}(t) = \exp\left(\int_0^t \mathbf{L}(\tau)\,\mathrm{d}\tau\right)$$

```
tau = linspace(0,t); d_tau = tau(2)-tau(1);
L_tau = (t-tau) .* L
F = expm(cumsum(L_tau * d_tau))
```

The Deformation Gradient Tensor **F**

describes the local deformation in continuum mechanics and satisfies

$$\dot{\mathbf{F}}(t) = \mathbf{L}(t)\,\mathbf{F}(t)$$

In case of constant deformation $\mathbf{L}(t) = \mathbf{L}$ we obtain

$$\mathbf{F}(t) = \exp(t \cdot \mathbf{L})$$

```
F = expm(L * t)
```

```
F = deformationGradientTensor (show methods, plot)
  rank: 2 (3 x 3)

 1 0 0
 2 1 0
 0 0 1
```

In the case of time dependent deformation we have

$$\mathbf{F}(t) = \exp\left(\int_0^t \mathbf{L}(\tau)\,\mathrm{d}\tau\right)$$

The Deformation Gradient Tensor **F**

describes the local deformation in continuum mechanics and satisfies

$$\dot{\mathbf{F}}(t) = \mathbf{L}(t)\,\mathbf{F}(t)$$

In case of constant deformation $\mathbf{L}(t) = \mathbf{L}$ we obtain

$$\mathbf{F}(t) = \exp(t \cdot \mathbf{L})$$

```
F = expm(L * t)
```

In the case of time dependent deformation we have

$$\mathbf{F}(t) = \exp\left(\int_0^t \mathbf{L}(\tau)\,\mathrm{d}\tau\right)$$

```
tau = linspace(0,t); d_tau = tau(2)-tau(1);
L_tau = (t-tau) .* L
F = expm(cumsum(L_tau * d_tau))
```

Decomposition of the Deformation Gradient Tensor

The deformation gradient tensor **F** can be decomposed into a rotational part **R**, a right stretch tensor **U** and a left stretch tensor **V** such that

$$\mathbf{F} = \mathbf{RU} = \mathbf{VR}$$

```
[R,V,U] = polar(F)
```

```
R = tensor (show methods, plot)
  rank: 2 (3 x 3)

  0.7071 -0.7071        0
  0.7071  0.7071        0
       0       0        1

U = tensor (show methods, plot)
  rank: 2 (3 x 3)

 2.1213 0.7071       0
 0.7071 0.7071       0
      0      0       1
```

Decomposition of the Deformation Gradient Tensor

The deformation gradient tensor **F** can be decomposed into a rotational part **R**, a right stretch tensor **U** and a left stretch tensor **V** such that

$$\mathbf{F} = \mathbf{RU} = \mathbf{VR}$$

```
[R,V,U] = polar(F)
```

The polar decomposition is closely related to the singular value decomposition of of **F**

```
[v,s,u] = svd(expm(L .* tau))
```

```
a = s(1); b = s(3);
plotEllipse([0,0],a,b,u(1).rho)
```

```
plotEllipse([0,0],a,b,v(1).rho)
```

Decomposition of the Deformation Gradient Tensor

The deformation gradient tensor **F** can be decomposed into a rotational part **R**, a right stretch tensor **U** and a left stretch tensor **V** such that

$$\mathbf{F} = \mathbf{RU} = \mathbf{VR}$$

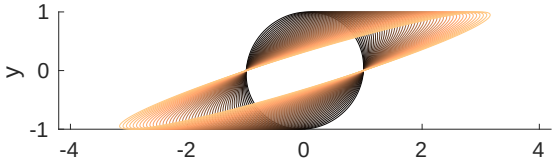
```
[R,V,U] = polar(F)
```

The polar decomposition is closely related to the singular value decomposition of of **F**

```
[v,s,u] = svd(expm(L .* tau))
```

```
a = s(1); b = s(3);
plotEllipse([0,0],a,b,u(1).rho)
```

```
plotEllipse([0,0],a,b,v(1).rho)
```



Decomposition of the Deformation Gradient Tensor

The deformation gradient tensor **F** can be decomposed into a rotational part **R**, a right stretch tensor **U** and a left stretch tensor **V** such that

$$\mathbf{F} = \mathbf{RU} = \mathbf{VR}$$

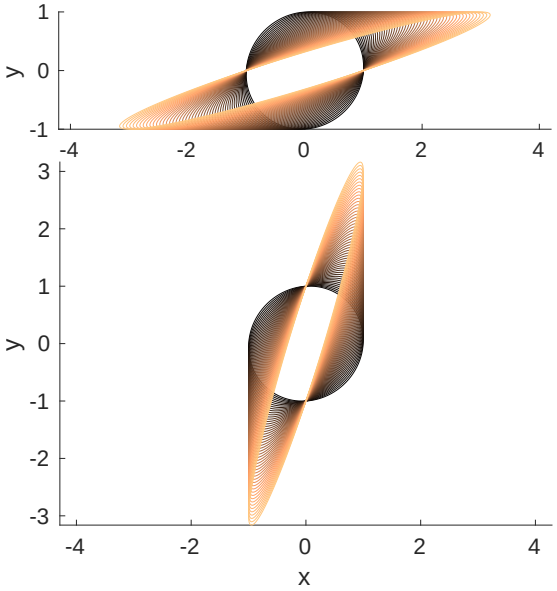
```
[R,V,U] = polar(F)
```

The polar decomposition is closely related to the singular value decomposition of of **F**

```
[v,s,u] = svd(expm(L .* tau))
```

```
a = s(1); b = s(3);
plotEllipse([0,0],a,b,u(1).rho)
```

```
plotEllipse([0,0],a,b,v(1).rho)
```



Strain Tensors

right Cauchy-Green stretch tensor $\mathbf{C} = \mathbf{U}^2 = \mathbf{F}'\mathbf{F}$

```
C = F' .* F
```

```
C = tensor (show methods, plot)
  rank: 2 (3 x 3)

      1 0.0524      0
 0.0524 1.0027      0
      0      0      1
```

Strain Tensors

right Cauchy-Green stretch tensor $\mathbf{C} = \mathbf{U}^2 = \mathbf{F}'\mathbf{F}$

```
C = F' .* F
```

left Cauchy-Green stretch tensor $\mathbf{B} = \mathbf{V}^2 = \mathbf{F}\mathbf{F}'$

```
B = F .* F'
```

```
B = tensor (show methods, plot)
  rank: 2 (3 x 3)

 1.0027 0.0524      0
 0.0524      1      0
      0      0      1
```

Strain Tensors

right Cauchy-Green stretch tensor $\mathbf{C} = \mathbf{U}^2 = \mathbf{F}'\mathbf{F}$

```
C = F' .* F
```

left Cauchy-Green stretch tensor $\mathbf{B} = \mathbf{V}^2 = \mathbf{F}\mathbf{F}'$

```
B = F .* F'
```

Green-Lagrange strain tensor

```
E = (C - tensor.eye)./2
```

```
E = tensor (show methods, plot)
  rank: 2 (3 x 3)

 *10^-3
      0 26.204       0
 26.204  1.373       0
      0       0       0
```

Strain Tensors

right Cauchy-Green stretch tensor $\mathbf{C} = \mathbf{U}^2 = \mathbf{F}'\mathbf{F}$

```
C = F' .* F
```

left Cauchy-Green stretch tensor $\mathbf{B} = \mathbf{V}^2 = \mathbf{F}\mathbf{F}'$

```
B = F .* F'
```

Green-Lagrange strain tensor

```
E = (C - tensor.eye)./2
```

Almansi strain tensor

```
e = (tensor.eye - inv(B))./2
```

```
e = tensor (show methods, plot)
  rank: 2 (3 x 3)

 *10^-3
      0 26.204       0
 26.204 -1.373       0
      0       0       0
```

Strain Tensors

right Cauchy-Green stretch tensor $\mathbf{C} = \mathbf{U}^2 = \mathbf{F}'\mathbf{F}$

```
C = F' .* F
```

left Cauchy-Green stretch tensor $\mathbf{B} = \mathbf{V}^2 = \mathbf{F}\mathbf{F}'$

```
B = F .* F'
```

Green-Lagrange strain tensor

```
E = (C - tensor.eye)./2
```

Almansi strain tensor

```
e = (tensor.eye - inv(B))./2
```

engineering strain

```
eps = F.sym - tensor.eye
```

```
eps = tensor (show methods, plot)
  rank: 2 (3 x 3)

 *10^-3
      0 26.204       0
 26.204       0       0
      0       0       0
```

Strain Tensors

right Cauchy-Green stretch tensor $\mathbf{C} = \mathbf{U}^2 = \mathbf{F}'\mathbf{F}$

```
C = F' .* F
```

left Cauchy-Green stretch tensor $\mathbf{B} = \mathbf{V}^2 = \mathbf{F}\mathbf{F}'$

```
B = F .* F'
```

Green-Lagrange strain tensor

```
E = (C - tensor.eye)./2
```

Almansi strain tensor

```
e = (tensor.eye - inv(B))./2
```

engineering strain

```
eps = F.sym - tensor.eye
```

true, logarithmic or Hencky strain tensor

```
H = logm(U)
```

```
H = tensor (show methods, plot)
  rank: 2 (3 x 3)

 *10^-3
 -0.686 26.192       0
 26.192  0.686       0
      0       0       0
```

Strain Tensors

right Cauchy-Green stretch tensor $\mathbf{C} = \mathbf{U}^2 = \mathbf{F}'\mathbf{F}$

```
C = F' .* F
```

left Cauchy-Green stretch tensor $\mathbf{B} = \mathbf{V}^2 = \mathbf{F}\mathbf{F}'$

```
B = F .* F'
```

Green-Lagrange strain tensor

```
E = (C - tensor.eye)./2
```

Almansi strain tensor

```
e = (tensor.eye - inv(B))./2
```

engineering strain

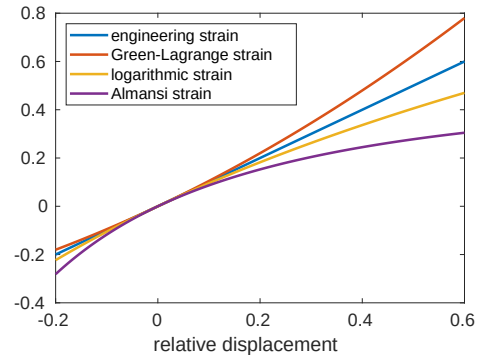
```
eps = F.sym - tensor.eye
```

true, logarithmic or Hencky strain tensor

```
H = logm(U)
```

```
H = tensor (show methods, plot)
  rank: 2 (3 x 3)

 *10^-3
 -0.686 26.192       0
 26.192  0.686       0
      0       0       0
```



Slip Systems

A slip system is specified by a normal direction **n** and a slip direction **d** orthogonal to **n**

```
d = Miller(0,-1,1,cs,'uvw');
n = Miller(1,1,1,cs,'hkl');
sS = slipSystem(d,n)
```

```
sS = slipSystem
 symmetry: 432
 CRSS: 1
 size: 1 x 1
  u   v   w | h   k   l
  0  -1   1   1   1   1
```

Slip Systems

A slip system is specified by a normal direction **n** and a slip direction **d** orthogonal to **n**

```
d = Miller(0,-1,1,cs,'uvw');
n = Miller(1,1,1,cs,'hkl');
sS = slipSystem(d,n)
```

build in fcc, bcc and hcp slip systems

```
sS = slipSystem.fcc(cs)
```

```
sS = slipSystem
 symmetry: 432
 CRSS: 1
 size: 1 x 1
  u   v   w | h   k   l
  0   1  -1   1   1   1
```

Slip Systems

A slip system is specified by a normal direction **n** and a slip direction **d** orthogonal to **n**

```
d = Miller(0,-1,1,cs,'uvw');
n = Miller(1,1,1,cs,'hkl');
sS = slipSystem(d,n)
```

build in fcc, bcc and hcp slip systems

```
sS = slipSystem.fcc(cs)
```

symmetrically equivalent slip systems

```
sS.symmetrise
```

```
sS = slipSystem
 symmetry: 432
 CRSS: 1
 size: 24 x 1
   u    v    w | h    k    l
   0    1   -1    1    1    1
  -1    0    1    1    1    1
   1   -1    0    1    1    1
   0   -1    1    1    1    1
   1    0   -1    1    1    1
  -1    1    0    1    1    1
  -1    1    0    1   -1    1
  -1    0   -1    1   -1    1
   0   -1   -1    1   -1    1
   1    0    1    1   -1    1
   0    1    1    1   -1    1
   1   -1    0    1   -1    1
   0    1   -1    1    1    1
   1    0    1    1    1    1
   1    1    0    1    1    1
  -1    0   -1    1    1    1
  -1   -1    0    1    1    1
   0   -1    1    1    1    1
  -1    0    1    1   -1    1
```

Slip Systems

A slip system is specified by a normal direction **n** and a slip direction **d** orthogonal to **n**

```
d = Miller(0,-1,1,cs,'uvw');
n = Miller(1,1,1,cs,'hkl');
sS = slipSystem(d,n)
```

build in fcc, bcc and hcp slip systems

```
sS = slipSystem.fcc(cs)
```

symmetrically equivalent slip systems

```
sS.symmetrise('antipodal')
```

```
sS = slipSystem
 symmetry: 432
 CRSS: 1
 size: 12 x 1
   u   v   w | h   k   l
   0  -1   1   1   1   1
   1   0  -1   1   1   1
  -1   1   0   1   1   1
  -1   1   0   1   1  -1
  -1   0  -1   1   1  -1
   0  -1  -1   1   1  -1
   0  -1   1  -1   1   1
  -1   0  -1  -1   1   1
  -1  -1   0  -1   1   1
   1   0  -1   1  -1   1
  -1  -1   0   1  -1   1
   0  -1  -1   1  -1   1
```

Slip Systems

A slip system is specified by a normal direction **n** and a slip direction **d** orthogonal to **n**

```
d = Miller(0,-1,1,cs,'uvw');
n = Miller(1,1,1,cs,'hkl');
sS = slipSystem(d,n)
```

build in fcc, bcc and hcp slip systems

```
sS = slipSystem.fcc(cs)
```

symmetrically equivalent slip systems

```
sS.symmetrise('antipodal')
```

transform to specimen reference frame

```
orientation.rand(cs) * sS
```

```
ans = slipSystem
 size: 1 x 1
     x      y      z  |     x      y      z
 -0.03  -1.03  -0.97    -1.38   0.74  -0.74
```

Slip Systems

A slip system is specified by a normal direction **n** and a slip direction **d** orthogonal to **n**

```
d = Miller(0,-1,1,cs,'uvw');
n = Miller(1,1,1,cs,'hkl');
sS = slipSystem(d,n)
```

build in fcc, bcc and hcp slip systems

```
sS = slipSystem.fcc(cs)
```

symmetrically equivalent slip systems

```
sS.symmetrise('antipodal')
```

transform to specimen reference frame

```
orientation.rand(cs) * sS
```

Schmid or deformation tensor

```
m = SchmidTensor(sS)
```

```
ans = velocityGradientTensor
  rank    : 2 (3 x 3)
  mineral: iron (m-3m)

 *10^-2
      0       0       0
  40.82   40.82   40.82
 -40.82  -40.82  -40.82
```

Schmid Factor

The Schmid factor is the resolved shear stress τ for a specific tension direction $\mathbf{r}$

$$\tau = \cos \measuredangle(\mathbf{r}, \mathbf{n}) \cdot \cos \measuredangle(\mathbf{r}, \mathbf{d}) = (\mathbf{r}^T \mathbf{n}) \cdot (\mathbf{r}^T \mathbf{d})$$

```
r = vector3d.Z
tau = dot(n,r) .* dot(d,r)
```

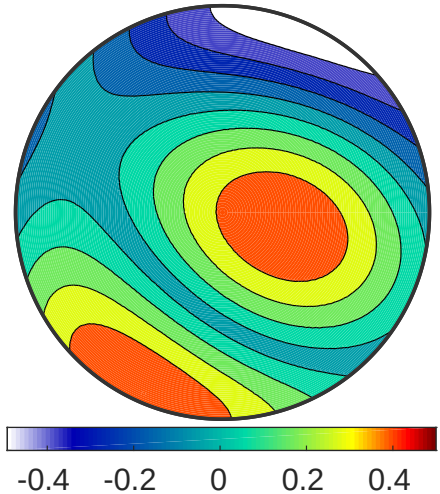
Schmid Factor

The Schmid factor is the resolved shear stress τ for a specific tension direction $\mathbf{r}$

$$\tau = \cos \measuredangle(\mathbf{r}, \mathbf{n}) \cdot \cos \measuredangle(\mathbf{r}, \mathbf{d}) = (\mathbf{r}^T \mathbf{n}) \cdot (\mathbf{r}^T \mathbf{d})$$

```
r = vector3d.Z
tau = dot(n,r) .* dot(d,r)
```

```
tau = sS.SchmidFactor(r)
plot(sS.SchmidFactor)
```



Schmid Factor

The Schmid factor is the resolved shear stress τ for a specific tension direction $\mathbf{r}$

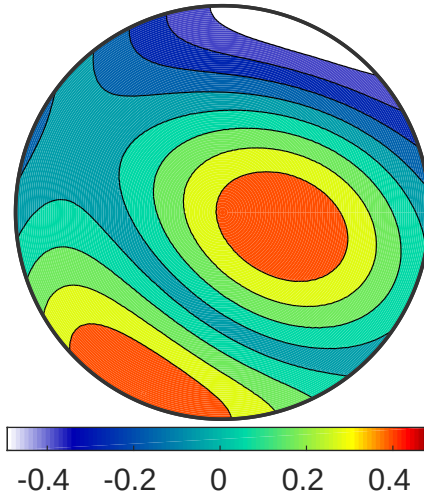
$$\tau = \cos \measuredangle(\mathbf{r}, \mathbf{n}) \cdot \cos \measuredangle(\mathbf{r}, \mathbf{d}) = (\mathbf{r}^T \mathbf{n}) \cdot (\mathbf{r}^T \mathbf{d})$$

```
r = vector3d.Z
tau = dot(n,r) .* dot(d,r)
```

```
tau = sS.SchmidFactor(r)
plot(sS.SchmidFactor)
```

For general stress σ we have $\tau = m : \sigma$

```
sigma = stressTensor.uniaxial(r)
tau = EinsteinSum(m,[-1 -2],...
                  sigma,[-1,-2])
tau = m : sigma
tau = sS.SchmidFactor(sigma)
```



Schmid Factor

The Schmid factor is the resolved shear stress τ for a specific tension direction $\mathbf{r}$

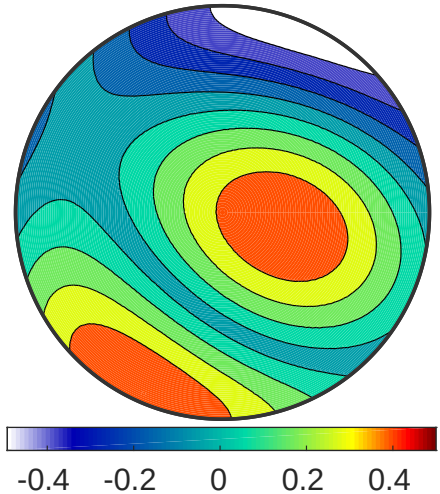
$$\tau = \cos \measuredangle(\mathbf{r}, \mathbf{n}) \cdot \cos \measuredangle(\mathbf{r}, \mathbf{d}) = (\mathbf{r}^T \mathbf{n}) \cdot (\mathbf{r}^T \mathbf{d})$$

```
r = vector3d.Z
tau = dot(n,r) .* dot(d,r)
```

```
tau = sS.SchmidFactor(r)
plot(sS.SchmidFactor)
```

For general stress σ we have $\tau = m : \sigma$

```
sigma = stressTensor.uniaxial(r)
tau = (ori * m) : sigma
tau = m : inv(ori) * sigma
```



Schmid Factor

The Schmid factor is the resolved shear stress τ for a specific tension direction $\mathbf{r}$

$$\tau = \cos \measuredangle(\mathbf{r}, \mathbf{n}) \cdot \cos \measuredangle(\mathbf{r}, \mathbf{d}) = (\mathbf{r}^T \mathbf{n}) \cdot (\mathbf{r}^T \mathbf{d})$$

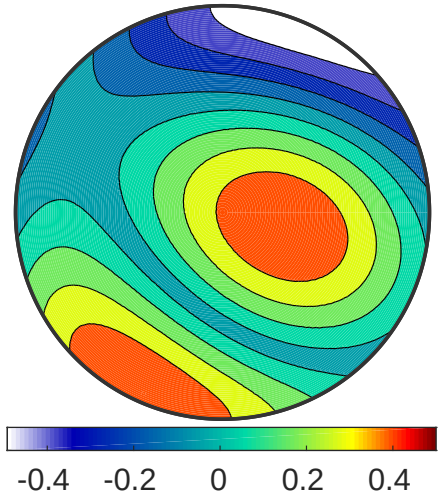
```
r = vector3d.Z
tau = dot(n,r) .* dot(d,r)
```

```
tau = sS.SchmidFactor(r)
plot(sS.SchmidFactor)
```

For general stress σ we have $\tau = m : \sigma$

```
sigma = stressTensor.uniaxial(r)
tau = (ori * m) : sigma
tau = m : inv(ori) * sigma
```

```
tau = SchmidFactor(ori*sS,sigma)
```



Maximum Schmid Factor

the Schmid factor for a list of tension directions results in a `length(r)` $\times$ `length(sS)` matrix `tau`

```
sS = sS.symmetrise('antipodal')
r = plotS2Grid('upper')
tau = sS.SchmidFactor(r)
```

maximum Schmidfactor over the second dimension

```
[tau_max,id] = max(abs(tau),[],2)
contourf(r,tau_max)
```

the active slip system

```
contourf(r,id,'contours',12)
hold on
quiver(r,sS(id).n,'color','r');
quiver(r,sS(id).b,'color','g');
hold off
```

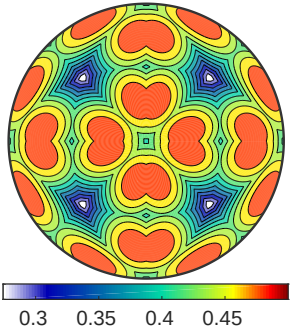
Maximum Schmid Factor

the Schmid factor for a list of tension directions results in a `length(r)` $\times$ `length(sS)` matrix `tau`

```
sS = sS.symmetrise('antipodal')
r = plotS2Grid('upper')
tau = sS.SchmidFactor(r)
```

maximum Schmidfactor over the second dimension

```
[tau_max,id] = max(abs(tau),[],2)
contourf(r,tau_max)
```



the active slip system

```
contourf(r,id,'contours',12)
hold on
quiver(r,sS(id).n,'color','r');
quiver(r,sS(id).b,'color','g');
hold off
```

Maximum Schmid Factor

the Schmid factor for a list of tension directions results in a `length(r)` $\times$ `length(sS)` matrix `tau`

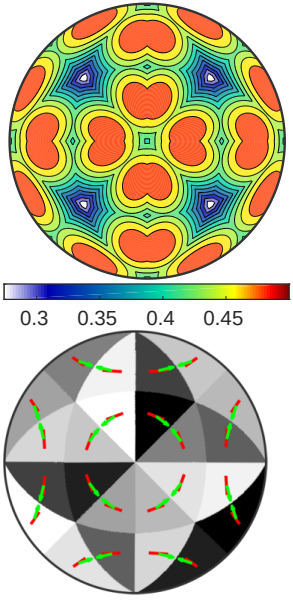
```
sS = sS.symmetrise('antipodal')
r = plotS2Grid('upper')
tau = sS.SchmidFactor(r)
```

maximum Schmidfactor over the second dimension

```
[tau_max,id] = max(abs(tau),[],2)
contourf(r,tau_max)
```

the active slip system

```
contourf(r,id,'contours',12)
hold on
quiver(r,sS(id).n,'color','r');
quiver(r,sS(id).b,'color','g');
hold off
```



Stress Based Analysis - Route 1

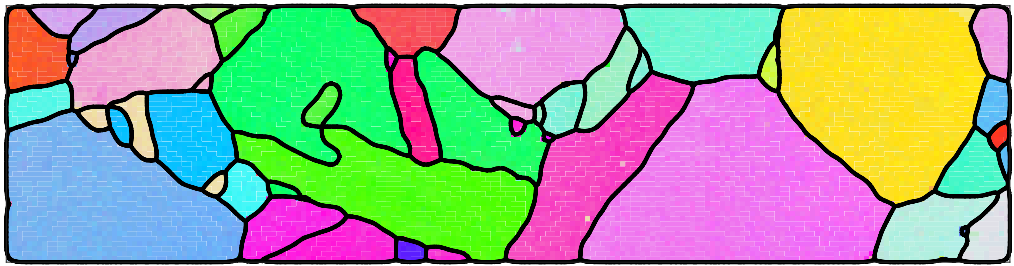
```
sS = symmetrise(slipSystem.fcc(ebsd.CS))
```

```
sS = slipSystem (show methods, plot)
 mineral: iron (m-3m)
 size: 24 x 1
   u    v    w | h    k    l
   0    1   -1    1    1    1
  -1    0    1    1    1    1
   1   -1    0    1    1    1
   0   -1    1    1    1    1
   1    0   -1    1    1    1
  -1    1    0    1    1    1
   1   -1    0    1    1   -1
   1    0    1    1    1   -1
   0    1    1    1    1   -1
  -1    0   -1    1    1   -1
   0   -1   -1    1    1   -1
  -1    1    0    1    1   -1
   0    1   -1   -1    1    1
   1    0    1   -1    1    1
   1    1    0   -1    1    1
  -1    0   -1   -1    1    1
  -1   -1    0   -1    1    1
   0   -1    1   -1    1    1
```

Stress Based Analysis - Route 1

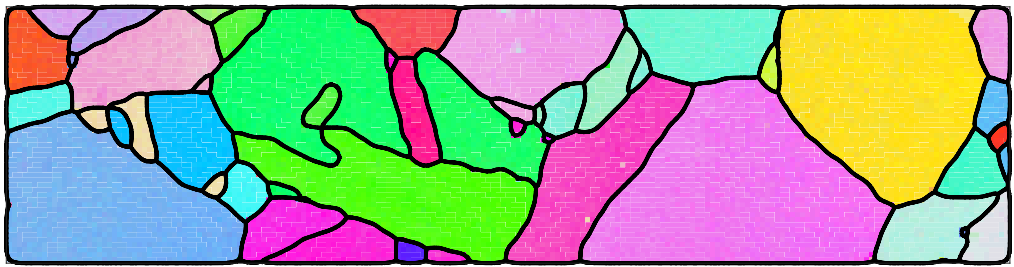
```
sS = symmetrise(slipSystem.fcc(ebsd.CS))
sSGrain = grains.meanOrientation .* sS
```

```
sSGrain = slipSystem (show methods, plot)
 size: 71 x 24
```



Stress Based Analysis - Route 1

```
sS = symmetrise(slipSystem.fcc(ebsd.CS))
sSGrain = grains.meanOrientation .* sS
SF = sSGrain.SchmidFactor(sigma);
```

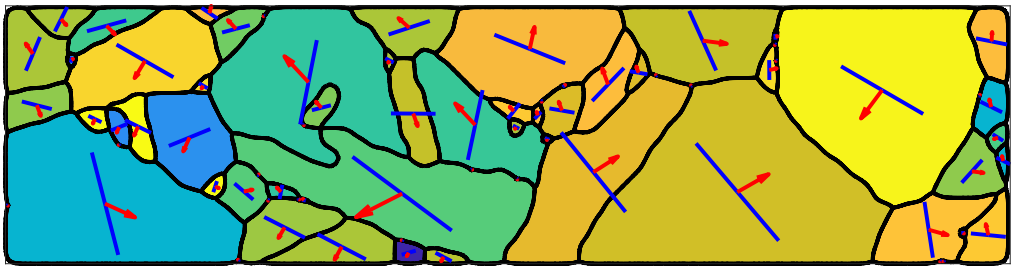


Stress Based Analysis - Route 1

```
sS = symmetrise(slipSystem.fcc(ebsd.CS))
sSGrain = grains.meanOrientation .* sS
SF = sSGrain.SchmidFactor(sigma);
[maxSF,active] = max(SF,[],2);
plot(grains,maxSF)
```

Stress Based Analysis - Route 1

```
sS = symmetrise(slipSystem.fcc(ebsd.CS))
sSGrain = grains.meanOrientation .* sS
SF = sSGrain.SchmidFactor(sigma);
[maxSF, active] = max(SF,[],2);
plot(grains,maxSF)
sSactive = grains.meanOrientation .* sS(active);
quiver(grains,sSactive.trace,'color','b')
quiver(grains,sSactive.b,'color','r')
```

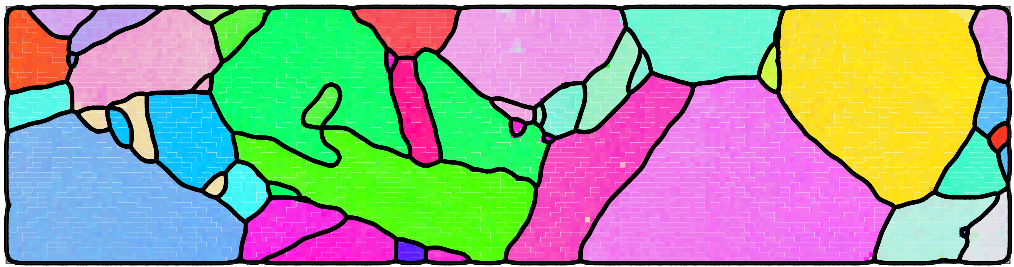


Stress Based Analysis - Route 2

```
sigma = stressTensor.uniaxial(vector3d.X)
```

```
sigma = stressTensor (show methods, plot)
  rank: 2 (3 x 3)

 1 0 0
 0 0 0
 0 0 0
```

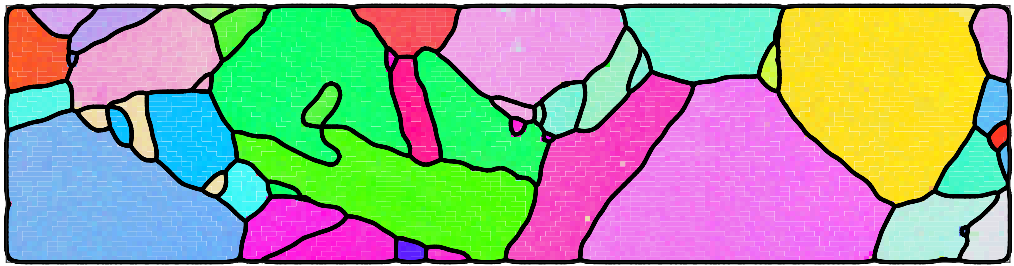


Stress Based Analysis - Route 2

```
sigma = stressTensor.uniaxial(vector3d.X)
```

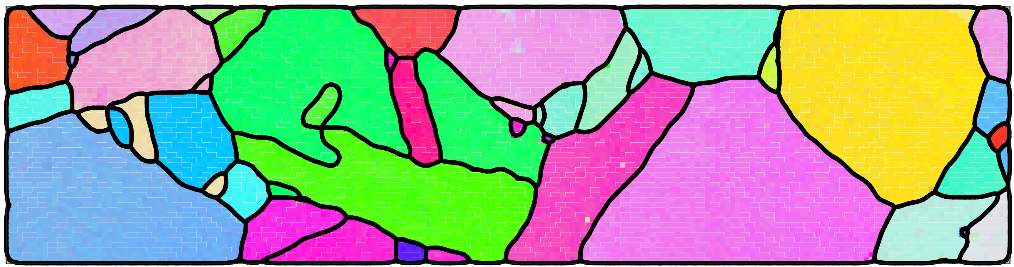
```
sigmaCrystal = inv(grains.meanOrientation) * sigma
```

```
sigmaCrystal = stressTensor (show methods, plot)
  size   : 71 x 1
  rank   : 2 (3 x 3)
  mineral: iron (m-3m)
```



Stress Based Analysis - Route 2

```
sigma = stressTensor.uniaxial(vector3d.X)
sigmaCrystal = inv(grains.meanOrientation) * sigma
SF = sS.SchmidFactor(sigmaCrystal);
```

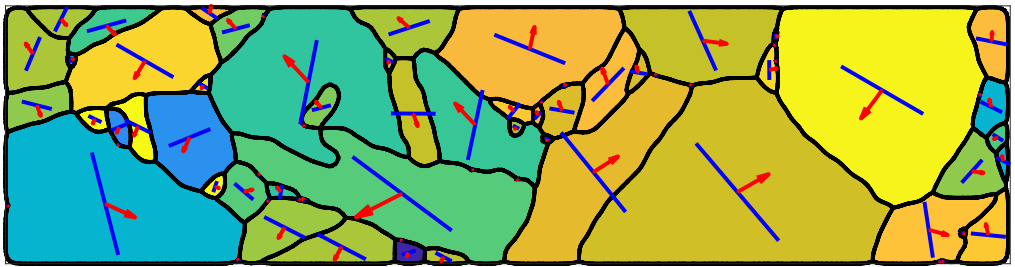


Stress Based Analysis - Route 2

```
sigma = stressTensor.uniaxial(vector3d.X)
sigmaCrystal = inv(grains.meanOrientation) * sigma
SF = sS.SchmidFactor(sigmaCrystal);
[maxSF,active] = max(abs(SF),[],2);
plot(grains,maxSF)
```

Stress Based Analysis - Route 2

```
sigma = stressTensor.uniaxial(vector3d.X)
sigmaCrystal = inv(grains.meanOrientation) * sigma
SF = sS.SchmidFactor(sigmaCrystal);
[maxSF, active] = max(abs(SF),[],2);
plot(grains,maxSF)
sSactive = grains.meanOrientation .* sS(active);
quiver(grains,sSactive.trace,'color','b')
quiver(grains,sSactive.b,'color','r')
```

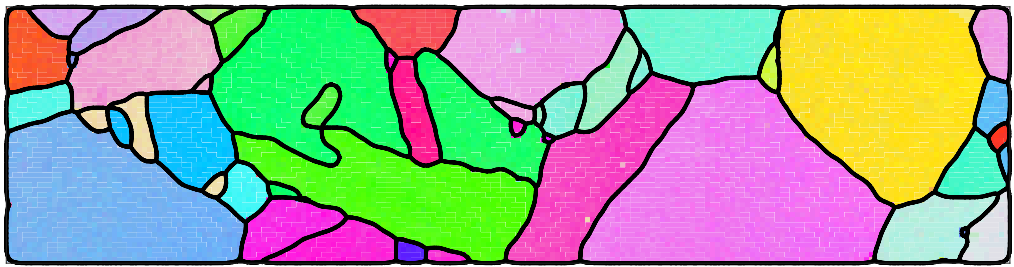


Strain Based Analysis

```
eps = strainTensor(diag([1,0,-1]))
```

```
sigma = strainTensor (show methods, plot)
  rank: 2 (3 x 3)

 1   0   0
 0   0   0
 0   0  -1
```

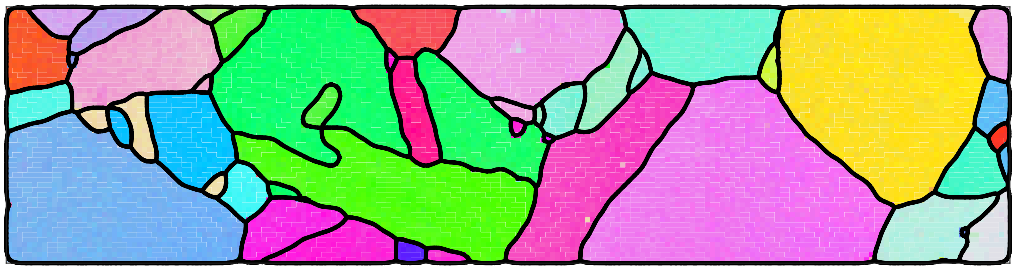


Strain Based Analysis

```
eps = strainTensor(diag([1,0,-1]))
```

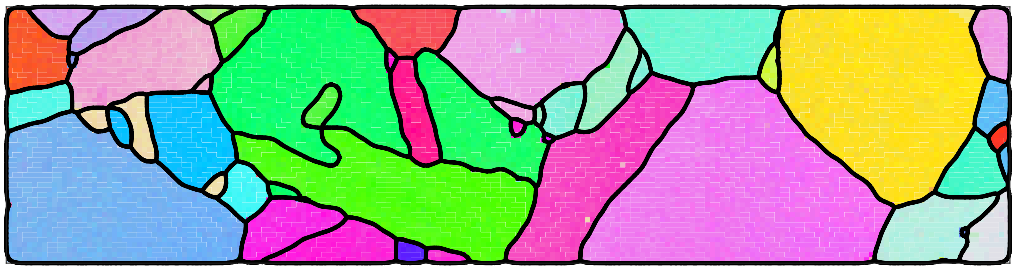
```
epsCrystal = inv(grains.meanOrientation) * eps
```

```
epsCrystal = strainTensor (show methods, plot)
  size    : 71 x 1
  rank    : 2 (3 x 3)
  mineral: iron (m-3m)
```



Strain Based Analysis

```
eps = strainTensor(diag([1,0,-1])
epsCrystal = inv(grains.meanOrientation) * eps
[TF, b, W] = calcTaylor(epsCrystal, sS);
```

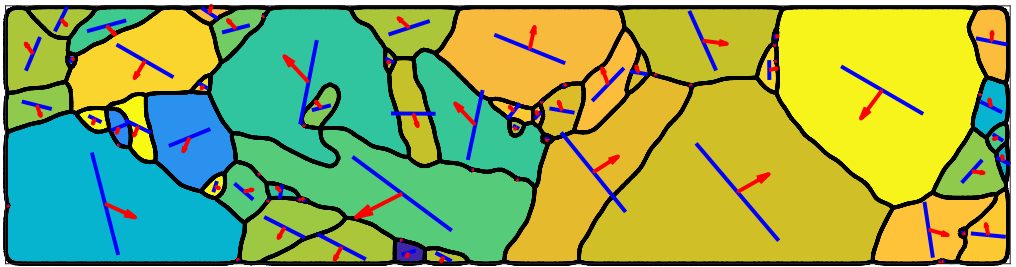


Strain Based Analysis

```
eps = strainTensor(diag([1,0,-1])
epsCrystal = inv(grains.meanOrientation) * eps
[TF, b, W] = calcTaylor(epsCrystal, sS);
plot(grains,TF)
```

Strain Based Analysis

```
eps = strainTensor(diag([1,0,-1])
epsCrystal = inv(grains.meanOrientation) * eps
[TF, b, W] = calcTaylor(epsCrystal, sS);
plot(grains,TF)
[bMax ,bMaxId] = max(b,[],2);
sSactive = grains.meanOrientation .* sS(bMaxId);
quiver(grains,sSactive.b,'color','red')
quiver(grains,sSactive.trace,'color','blue')
```



Rolling texture evolution

```
ori = orientation.rand(10000,CS)
h = Miller({0,0,1},{1,1,1},CS)
plotPDF(ori,'contourf')
```

the slip systems

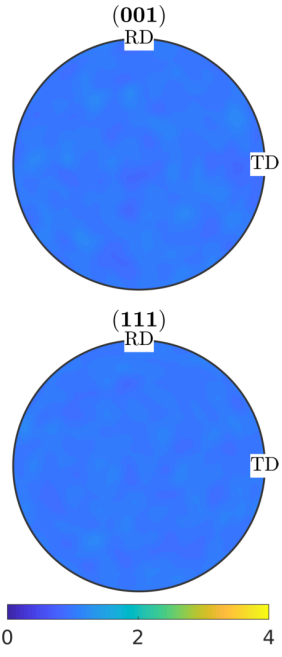
```
sS=symmetrise(slipSystem.fcc(CS))
```

30 percent strain

```
eps = strainTensor(diag([0.3 0 -0.3])) / 10
```

iterate deformation:

```
for i = 1:10
  [TF,b,W] = calcTaylor( inv(ori) * eps, sS);
  ori = ori .* orientation(-W);
  plotPDF(ori,h,'contourf')
end
```



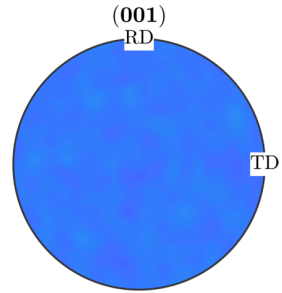
Rolling texture evolution

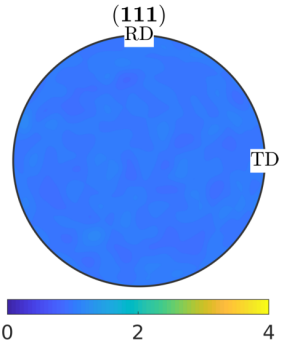
```
ori = orientation.rand(10000,CS)
h = Miller({0,0,1},{1,1,1}),CS)
plotPDF(ori,'contourf')
```

the slip systems

```
sS=symmetrise(slipSystem.fcc(CS))
```

```
sS = slipSystem (show methods, plot)
 size: 12 x 1
 mineral: Austenite (fcc) (432)
   u    v    w | h    k    l
   0    1   -1    1    1    1
  -1    0    1    1    1    1
   1   -1    0    1    1    1
   1   -1    0    1    1   -1
   1    0    1    1    1   -1
   0    1    1    1    1   -1
   0    1   -1   -1    1    1
   1    0    1   -1    1    1
   1    1    0   -1    1    1
  -1    0    1    1   -1    1
```





Rolling texture evolution

```
ori = orientation.rand(10000,CS)
h = Miller({0,0,1},{1,1,1},CS)
plotPDF(ori,'contourf')
```

the slip systems

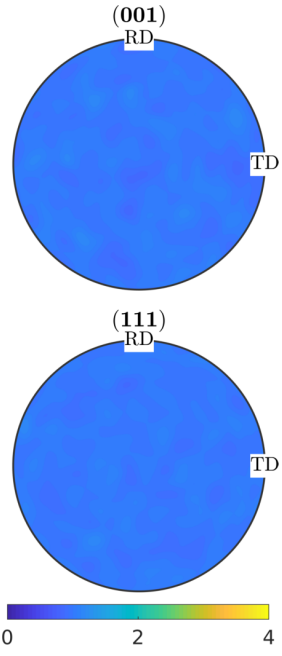
```
sS=symmetrise(slipSystem.fcc(CS))
```

30 percent strain

```
eps = strainTensor(diag([0.3 0 -0.3])) / 10
```

iterate deformation:

```
for i = 1:10
  [TF,b,W] = calcTaylor( inv(ori) * eps, sS);
  ori = ori .* orientation(-W);
  plotPDF(ori,h,'contourf')
end
```



Rolling texture evolution

```
ori = orientation.rand(10000,CS)
h = Miller({0,0,1},{1,1,1}),CS)
plotPDF(ori,'contourf')
```

the slip systems

```
sS=symmetrise(slipSystem.fcc(CS))
```

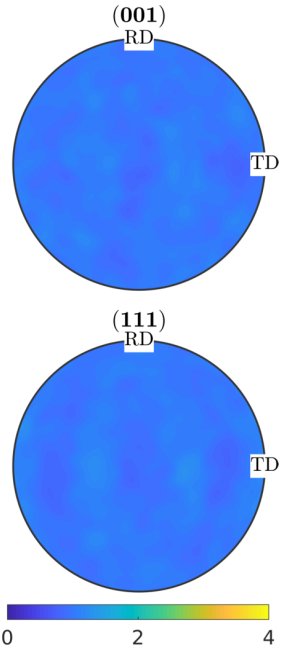
30 percent strain

```
eps = strainTensor(diag([0.3 0 -0.3])) / 10
```

iterate deformation:

```
for i = 1:10
  [TF,b,W] = calcTaylor( inv(ori) * eps, sS);
  ori = ori .* orientation(-W);
  plotPDF(ori,h,'contourf')
end
```

i=1



Rolling texture evolution

```
ori = orientation.rand(10000,CS)
h = Miller({0,0,1},{1,1,1},CS)
plotPDF(ori,'contourf')
```

the slip systems

```
sS=symmetrise(slipSystem.fcc(CS))
```

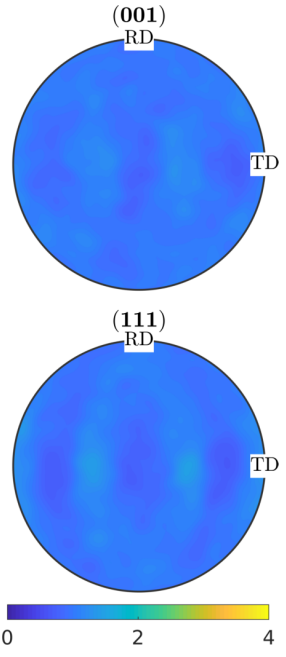
30 percent strain

```
eps = strainTensor(diag([0.3 0 -0.3])) / 10
```

iterate deformation:

```
for i = 1:10
  [TF,b,W] = calcTaylor( inv(ori) * eps, sS);
  ori = ori .* orientation(-W);
  plotPDF(ori,h,'contourf')
end
```

i=2



Rolling texture evolution

```
ori = orientation.rand(10000,CS)
h = Miller({0,0,1},{1,1,1},CS)
plotPDF(ori,'contourf')
```

the slip systems

```
sS=symmetrise(slipSystem.fcc(CS))
```

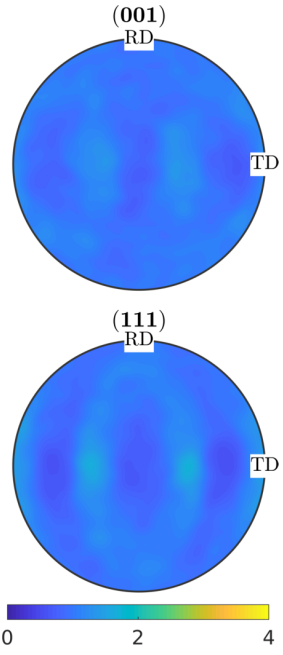
30 percent strain

```
eps = strainTensor(diag([0.3 0 -0.3])) / 10
```

iterate deformation:

```
for i = 1:10
  [TF,b,W] = calcTaylor( inv(ori) * eps, sS);
  ori = ori .* orientation(-W);
  plotPDF(ori,h,'contourf')
end
```

i=3



Rolling texture evolution

```
ori = orientation.rand(10000,CS)
h = Miller({0,0,1},{1,1,1},CS)
plotPDF(ori,'contourf')
```

the slip systems

```
sS=symmetrise(slipSystem.fcc(CS))
```

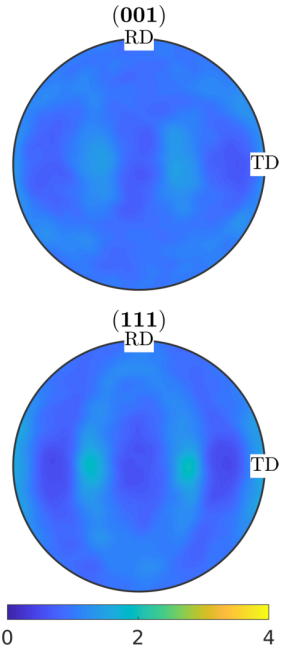
30 percent strain

```
eps = strainTensor(diag([0.3 0 -0.3])) / 10
```

iterate deformation:

```
for i = 1:10
  [TF,b,W] = calcTaylor( inv(ori) * eps, sS);
  ori = ori .* orientation(-W);
  plotPDF(ori,h,'contourf')
end
```

i=4



Rolling texture evolution

```
ori = orientation.rand(10000,CS)
h = Miller({0,0,1},{1,1,1},CS)
plotPDF(ori,'contourf')
```

the slip systems

```
sS=symmetrise(slipSystem.fcc(CS))
```

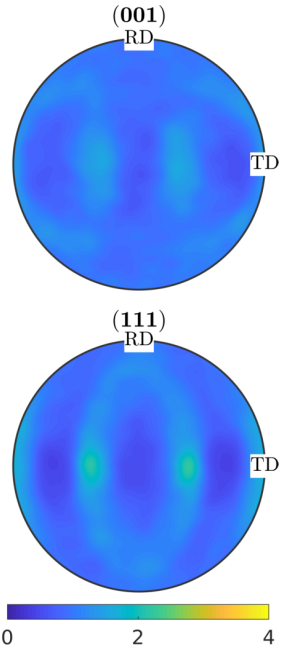
30 percent strain

```
eps = strainTensor(diag([0.3 0 -0.3])) / 10
```

iterate deformation:

```
for i = 1:10
  [TF,b,W] = calcTaylor( inv(ori) * eps, sS);
  ori = ori .* orientation(-W);
  plotPDF(ori,h,'contourf')
end
```

i=5



Rolling texture evolution

```
ori = orientation.rand(10000,CS)
h = Miller({0,0,1},{1,1,1},CS)
plotPDF(ori,'contourf')
```

the slip systems

```
sS=symmetrise(slipSystem.fcc(CS))
```

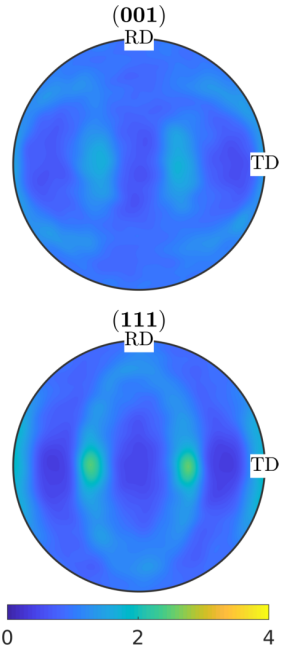
30 percent strain

```
eps = strainTensor(diag([0.3 0 -0.3])) / 10
```

iterate deformation:

```
for i = 1:10
  [TF,b,W] = calcTaylor( inv(ori) * eps, sS);
  ori = ori .* orientation(-W);
  plotPDF(ori,h,'contourf')
end
```

i=6



Rolling texture evolution

```
ori = orientation.rand(10000,CS)
h = Miller({0,0,1},{1,1,1},CS)
plotPDF(ori,'contourf')
```

the slip systems

```
sS=symmetrise(slipSystem.fcc(CS))
```

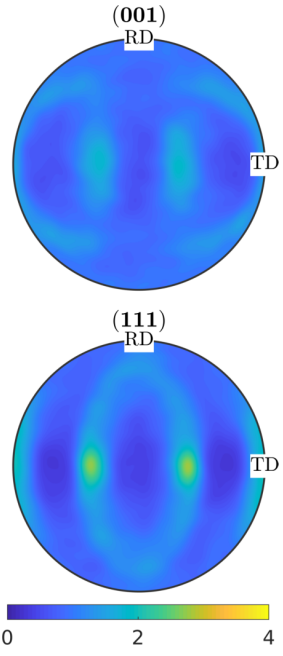
30 percent strain

```
eps = strainTensor(diag([0.3 0 -0.3])) / 10
```

iterate deformation:

```
for i = 1:10
  [TF,b,W] = calcTaylor( inv(ori) * eps, sS);
  ori = ori .* orientation(-W);
  plotPDF(ori,h,'contourf')
end
```

i=7



Rolling texture evolution

```
ori = orientation.rand(10000,CS)
h = Miller({0,0,1},{1,1,1},CS)
plotPDF(ori,'contourf')
```

the slip systems

```
sS=symmetrise(slipSystem.fcc(CS))
```

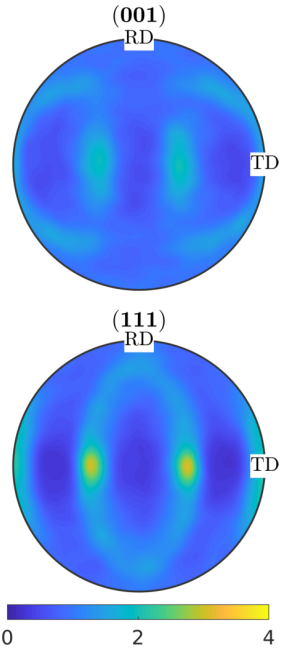
30 percent strain

```
eps = strainTensor(diag([0.3 0 -0.3])) / 10
```

iterate deformation:

```
for i = 1:10
  [TF,b,W] = calcTaylor( inv(ori) * eps, sS);
  ori = ori .* orientation(-W);
  plotPDF(ori,h,'contourf')
end
```

i=8



Rolling texture evolution

```
ori = orientation.rand(10000,CS)
h = Miller({0,0,1},{1,1,1},CS)
plotPDF(ori,'contourf')
```

the slip systems

```
sS=symmetrise(slipSystem.fcc(CS))
```

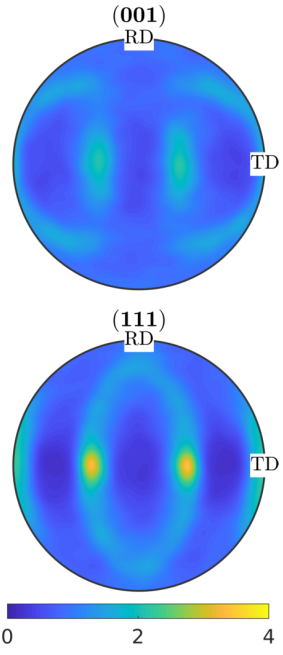
30 percent strain

```
eps = strainTensor(diag([0.3 0 -0.3])) / 10
```

iterate deformation:

```
for i = 1:10
  [TF,b,W] = calcTaylor( inv(ori) * eps, sS);
  ori = ori .* orientation(-W);
  plotPDF(ori,h,'contourf')
end
```

i=9



Rolling texture evolution

```
ori = orientation.rand(10000,CS)
h = Miller({0,0,1},{1,1,1},CS)
plotPDF(ori,'contourf')
```

the slip systems

```
sS=symmetrise(slipSystem.fcc(CS))
```

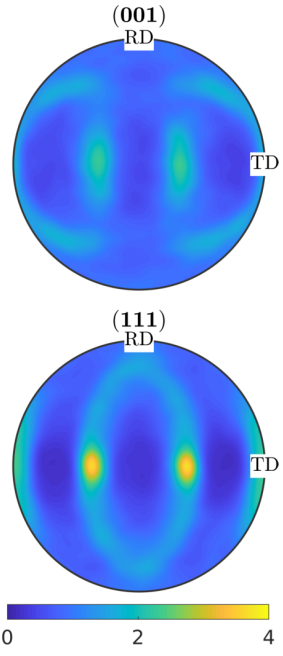
30 percent strain

```
eps = strainTensor(diag([0.3 0 -0.3])) / 10
```

iterate deformation:

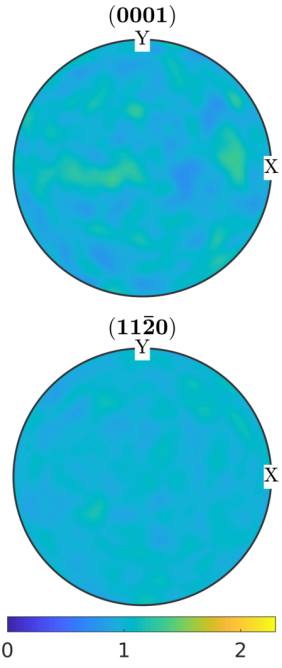
```
for i = 1:10
  [TF,b,W] = calcTaylor( inv(ori) * eps, sS);
  ori = ori .* orientation(-W);
  plotPDF(ori,h,'contourf')
end
```

i=10



Texture Evolution During Simple Shear

```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```



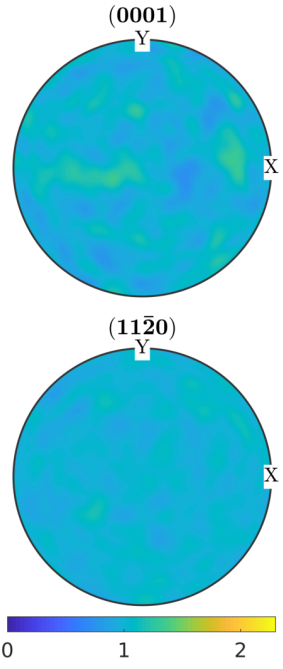
Texture Evolution During Simple Shear

```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```

```
F = deformationGradientTensor.simpleShear(...
    vector3d.Y, vector3d.X, 70*degree)
```

```
F = deformationGradientTensor
  rank: 2 (3 x 3)

       1       0       0
  2.7475       1       0
       0       0       1
```



Texture Evolution During Simple Shear

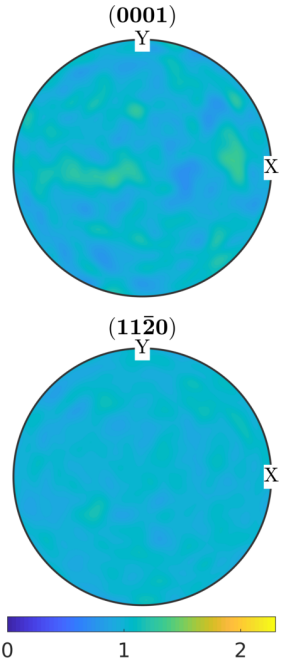
```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```

```
F = deformationGradientTensor.simpleShear(...
    vector3d.Y, vector3d.X, 70*degree)
```

```
L = logm(F) ./ nSteps
W = L.antiSym, D = L.sym
```

```
W = spinTensor                    D = strainRateTensor
  rank: 2 (3 x 3)                   rank: 2 (3 x 3)

 *10^-3                            *10^-3
   0 -30    0                        0 30   0
  30   0    0                       30  0   0
   0   0    0                        0  0   0
```



Texture Evolution During Simple Shear

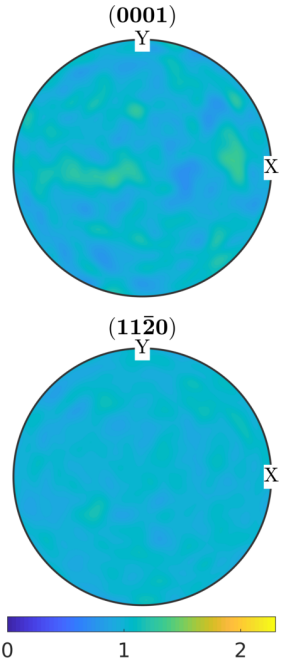
```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```

```
F = deformationGradientTensor.simpleShear(...
    vector3d.Y, vector3d.X, 70*degree)
```

```
L = logm(F) ./ nSteps
W = L.antiSym, D = L.sym
```

```
[~,~,W_Taylor] = calcTaylor(inv(ori) .* D, sS)
```

```
W_Taylor = spinTensor
  size    : 10000 x 1
  rank    : 2 (3 x 3)
  mineral: -3m1, X||a*, Y||b, Z||c*
```



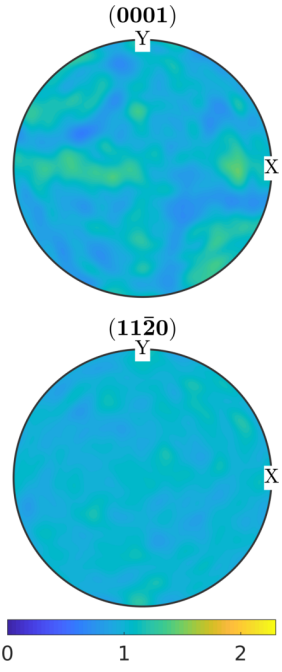
Texture Evolution During Simple Shear

```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```

```
F = deformationGradientTensor.simpleShear(...
    vector3d.Y, vector3d.X, 70*degree)
```

```
L = logm(F) ./ nSteps
W = L.antiSym, D = L.sym
```

```
for i = 1:nSteps
  [~,~,W_Taylor] = calcTaylor(inv(ori) * D, sS)
  ori = rotation(-W) .* ori
        .* orientation(-W_Taylor)
end
```



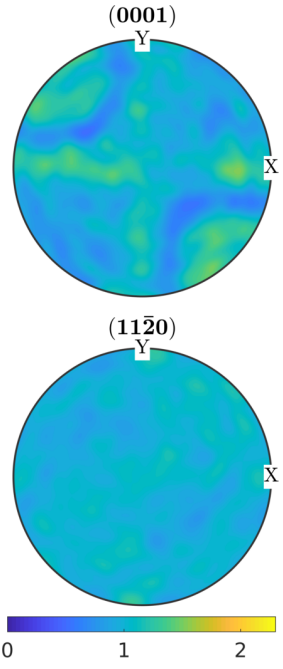
Texture Evolution During Simple Shear

```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```

```
F = deformationGradientTensor.simpleShear(...
    vector3d.Y, vector3d.X, 70*degree)
```

```
L = logm(F) ./ nSteps
W = L.antiSym, D = L.sym
```

```
for i = 1:nSteps
  [~,~,W_Taylor] = calcTaylor(inv(ori) * D, sS)
  W_total = W + ori .* W_Taylor;
  ori = rotation(-W_total) .* ori;
end
```



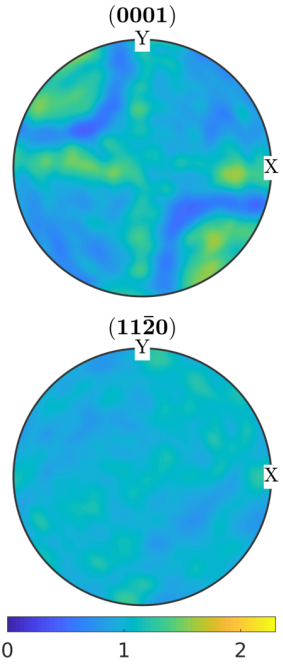
Texture Evolution During Simple Shear

```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```

```
F = deformationGradientTensor.simpleShear(...
    vector3d.Y, vector3d.X, 70*degree)
```

```
L = logm(F) ./ nSteps
W = L.antiSym, D = L.sym
```

```
for i = 1:nSteps
  [~,~,W_Taylor] = calcTaylor(inv(ori) * D, sS)
  W_total = inv(ori) * W + spin;
  ori = ori .* orientation(-W_total);
end
```



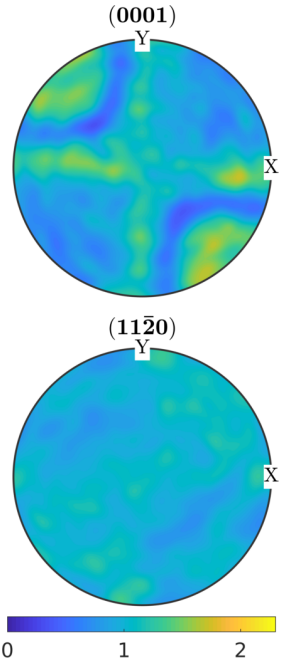
Texture Evolution During Simple Shear

```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```

```
F = deformationGradientTensor.simpleShear(...
    vector3d.Y, vector3d.X, 70*degree)
```

```
L = logm(F) ./ nSteps
W = L.antiSym, D = L.sym
```

```
for i = 1:nSteps
  [~,~,W_Taylor] = calcTaylor(inv(ori) * D, sS)
  W_total = inv(ori) * W + spin;
  ori = ori .* orientation(-W_total);
end
```



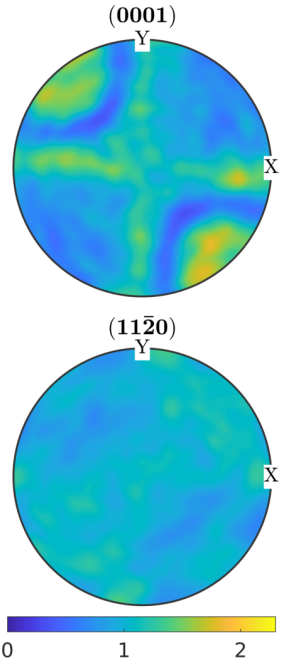
Texture Evolution During Simple Shear

```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```

```
F = deformationGradientTensor.simpleShear(...
    vector3d.Y, vector3d.X, 70*degree)
```

```
L = logm(F) ./ nSteps
W = L.antiSym, D = L.sym
```

```
for i = 1:nSteps
  [~,~,W_Taylor] = calcTaylor(inv(ori) * D, sS)
  W_total = inv(ori) * W + spin;
  ori = ori .* orientation(-W_total);
end
```



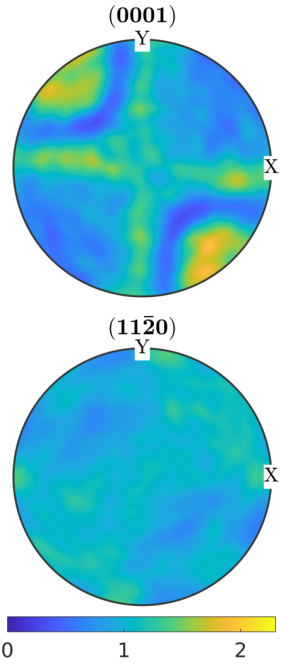
Texture Evolution During Simple Shear

```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```

```
F = deformationGradientTensor.simpleShear(...
    vector3d.Y, vector3d.X, 70*degree)
```

```
L = logm(F) ./ nSteps
W = L.antiSym, D = L.sym
```

```
for i = 1:nSteps
  [~,~,W_Taylor] = calcTaylor(inv(ori) * D, sS)
  W_total = inv(ori) * W + spin;
  ori = ori .* orientation(-W_total);
end
```



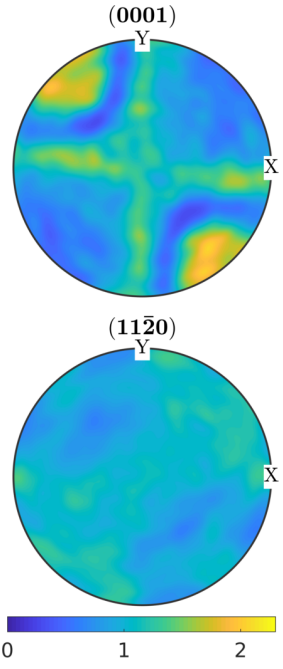
Texture Evolution During Simple Shear

```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```

```
F = deformationGradientTensor.simpleShear(...
    vector3d.Y, vector3d.X, 70*degree)
```

```
L = logm(F) ./ nSteps
W = L.antiSym, D = L.sym
```

```
for i = 1:nSteps
  [~,~,W_Taylor] = calcTaylor(inv(ori) * D, sS)
  W_total = inv(ori) * W + spin;
  ori = ori .* orientation(-W_total);
end
```



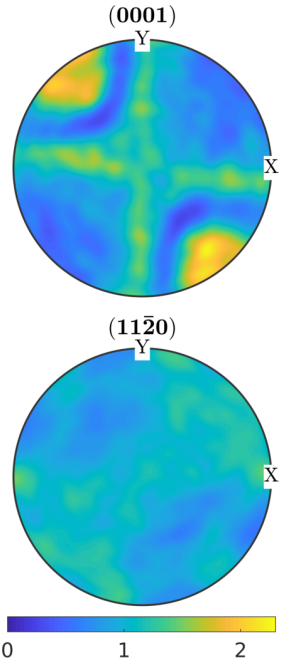
Texture Evolution During Simple Shear

```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```

```
F = deformationGradientTensor.simpleShear(...
    vector3d.Y, vector3d.X, 70*degree)
```

```
L = logm(F) ./ nSteps
W = L.antiSym, D = L.sym
```

```
for i = 1:nSteps
  [~,~,W_Taylor] = calcTaylor(inv(ori) * D, sS)
  W_total = inv(ori) * W + spin;
  ori = ori .* orientation(-W_total);
end
```



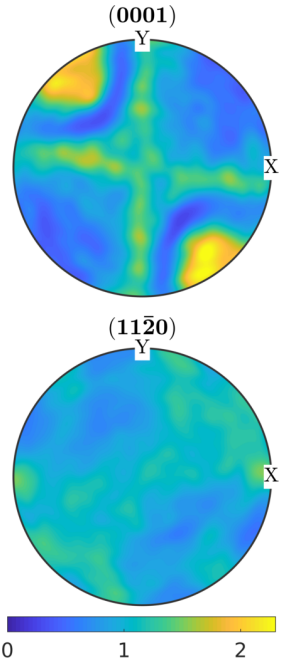
Texture Evolution During Simple Shear

```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```

```
F = deformationGradientTensor.simpleShear(...
    vector3d.Y, vector3d.X, 70*degree)
```

```
L = logm(F) ./ nSteps
W = L.antiSym, D = L.sym
```

```
for i = 1:nSteps
  [~,~,W_Taylor] = calcTaylor(inv(ori) * D, sS)
  W_total = inv(ori) * W + spin;
  ori = ori .* orientation(-W_total);
end
```



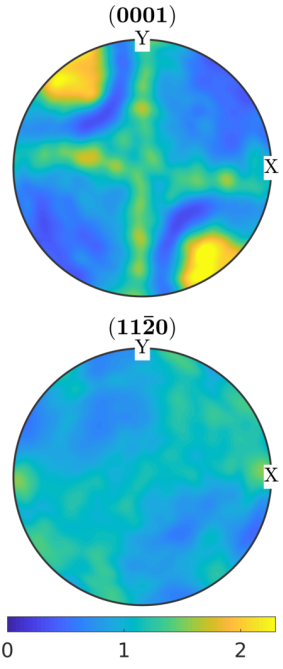
Texture Evolution During Simple Shear

```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```

```
F = deformationGradientTensor.simpleShear(...
    vector3d.Y, vector3d.X, 70*degree)
```

```
L = logm(F) ./ nSteps
W = L.antiSym, D = L.sym
```

```
for i = 1:nSteps
  [~,~,W_Taylor] = calcTaylor(inv(ori) * D, sS)
  W_total = inv(ori) * W + spin;
  ori = ori .* orientation(-W_total);
end
```



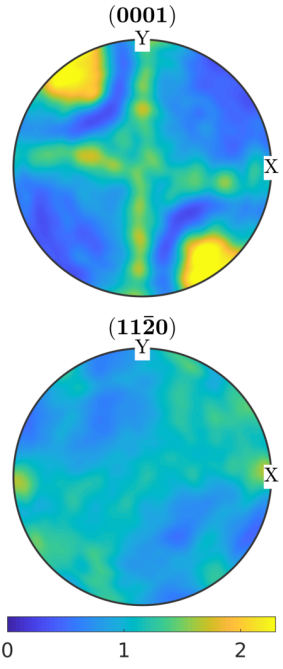
Texture Evolution During Simple Shear

```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```

```
F = deformationGradientTensor.simpleShear(...
    vector3d.Y, vector3d.X, 70*degree)
```

```
L = logm(F) ./ nSteps
W = L.antiSym, D = L.sym
```

```
for i = 1:nSteps
  [~,~,W_Taylor] = calcTaylor(inv(ori) * D, sS)
  W_total = inv(ori) * W + spin;
  ori = ori .* orientation(-W_total);
end
```



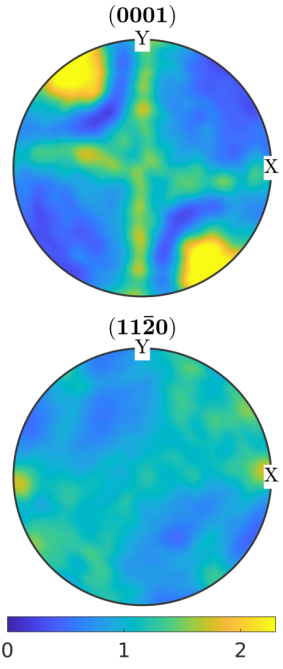
Texture Evolution During Simple Shear

```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```

```
F = deformationGradientTensor.simpleShear(...
    vector3d.Y, vector3d.X, 70*degree)
```

```
L = logm(F) ./ nSteps
W = L.antiSym, D = L.sym
```

```
for i = 1:nSteps
  [~,~,W_Taylor] = calcTaylor(inv(ori) * D, sS)
  W_total = inv(ori) * W + spin;
  ori = ori .* orientation(-W_total);
end
```



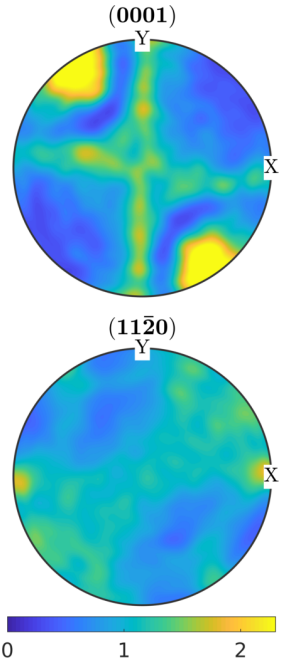
Texture Evolution During Simple Shear

```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```

```
F = deformationGradientTensor.simpleShear(...
    vector3d.Y, vector3d.X, 70*degree)
```

```
L = logm(F) ./ nSteps
W = L.antiSym, D = L.sym
```

```
for i = 1:nSteps
  [~,~,W_Taylor] = calcTaylor(inv(ori) * D, sS)
  W_total = inv(ori) * W + spin;
  ori = ori .* orientation(-W_total);
end
```



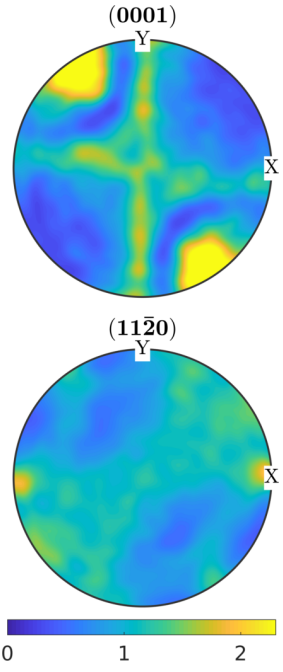
Texture Evolution During Simple Shear

```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```

```
F = deformationGradientTensor.simpleShear(...
    vector3d.Y, vector3d.X, 70*degree)
```

```
L = logm(F) ./ nSteps
W = L.antiSym, D = L.sym
```

```
for i = 1:nSteps
  [~,~,W_Taylor] = calcTaylor(inv(ori) * D, sS)
  W_total = inv(ori) * W + spin;
  ori = ori .* orientation(-W_total);
end
```



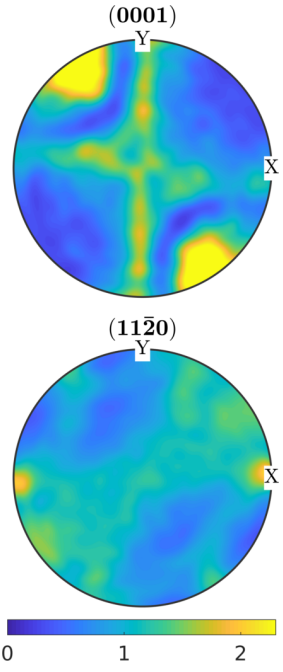
Texture Evolution During Simple Shear

```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```

```
F = deformationGradientTensor.simpleShear(...
    vector3d.Y, vector3d.X, 70*degree)
```

```
L = logm(F) ./ nSteps
W = L.antiSym, D = L.sym
```

```
for i = 1:nSteps
  [~,~,W_Taylor] = calcTaylor(inv(ori) * D, sS)
  W_total = inv(ori) * W + spin;
  ori = ori .* orientation(-W_total);
end
```



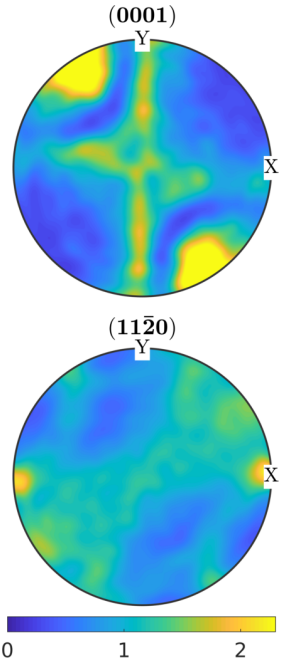
Texture Evolution During Simple Shear

```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```

```
F = deformationGradientTensor.simpleShear(...
    vector3d.Y, vector3d.X, 70*degree)
```

```
L = logm(F) ./ nSteps
W = L.antiSym, D = L.sym
```

```
for i = 1:nSteps
  [~,~,W_Taylor] = calcTaylor(inv(ori) * D, sS)
  W_total = inv(ori) * W + spin;
  ori = ori .* orientation(-W_total);
end
```



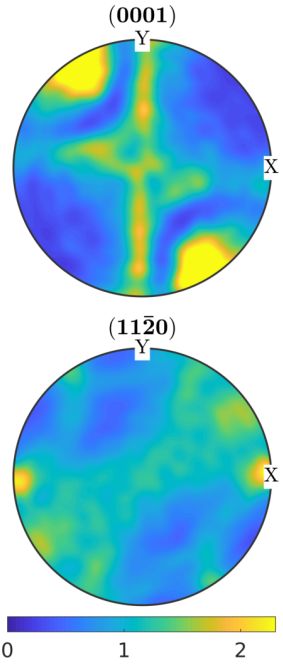
Texture Evolution During Simple Shear

```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```

```
F = deformationGradientTensor.simpleShear(...
    vector3d.Y, vector3d.X, 70*degree)
```

```
L = logm(F) ./ nSteps
W = L.antiSym, D = L.sym
```

```
for i = 1:nSteps
  [~,~,W_Taylor] = calcTaylor(inv(ori) * D, sS)
  W_total = inv(ori) * W + spin;
  ori = ori .* orientation(-W_total);
end
```



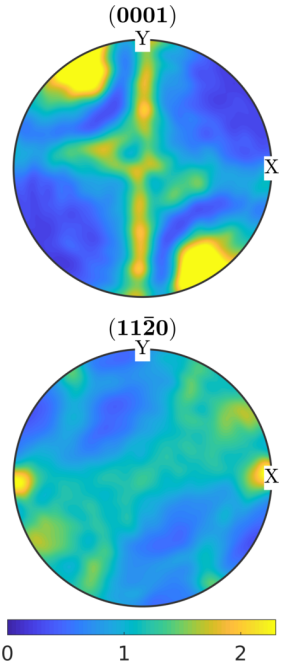
Texture Evolution During Simple Shear

```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```

```
F = deformationGradientTensor.simpleShear(...
    vector3d.Y, vector3d.X, 70*degree)
```

```
L = logm(F) ./ nSteps
W = L.antiSym, D = L.sym
```

```
for i = 1:nSteps
  [~,~,W_Taylor] = calcTaylor(inv(ori) * D, sS)
  W_total = inv(ori) * W + spin;
  ori = ori .* orientation(-W_total);
end
```



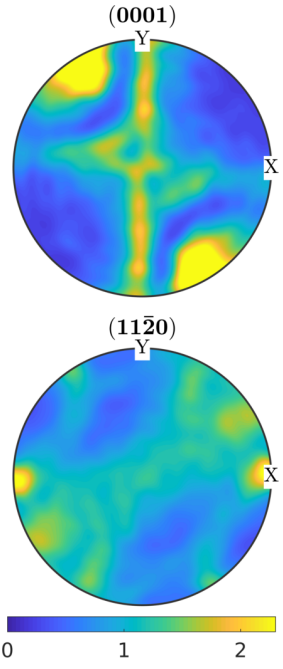
Texture Evolution During Simple Shear

```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```

```
F = deformationGradientTensor.simpleShear(...
    vector3d.Y, vector3d.X, 70*degree)
```

```
L = logm(F) ./ nSteps
W = L.antiSym, D = L.sym
```

```
for i = 1:nSteps
  [~,~,W_Taylor] = calcTaylor(inv(ori) * D, sS)
  W_total = inv(ori) * W + spin;
  ori = ori .* orientation(-W_total);
end
```



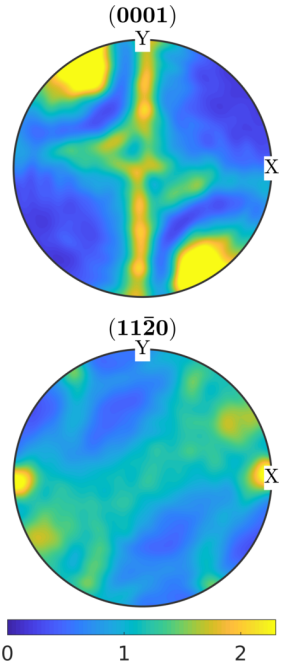
Texture Evolution During Simple Shear

```
cs = crystalSymmetry('-3m');
sS = [slipSystem({1,1,-2,0},{0,0,0,1},15,cs); ...
  slipSystem({0,0,0,1},{1,0,-1,0},30,cs); ...
  slipSystem({2,-1,-1,0},{0,1,-1,1},1,cs); ...
  slipSystem({2,-1,-1,0},{-1,0,1,0},1,cs)];
```

```
F = deformationGradientTensor.simpleShear(...
    vector3d.Y, vector3d.X, 70*degree)
```

```
L = logm(F) ./ nSteps
W = L.antiSym, D = L.sym
```

```
for i = 1:nSteps
  [~,~,W_Taylor] = calcTaylor(inv(ori) * D, sS)
  W_total = inv(ori) * W + spin;
  ori = ori .* orientation(-W_total);
end
```



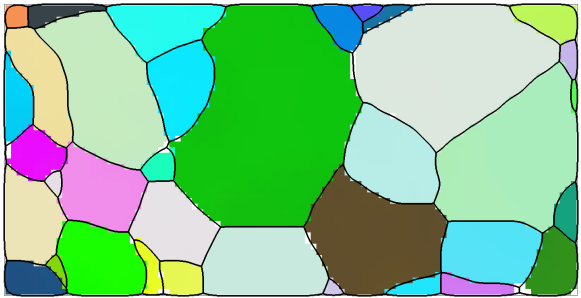
Geometrically Necessary Dislocations

```
ebsd = loadEBSD('DC06_2undef.ang')
```

```
ebsd = EBSD (show methods, plot)

 Phase  Orientations        Mineral  Symmetry
     0          5123  Iron (Alpha)       432

 Properties: ci, fit, iq, sem_signal, x, y, grainId
 Scan unit : um
 Grid size : 51 x 101
```



Geometrically Necessary Dislocations

```
ebsd = loadEBSD('DC06_2undef.ang')
```

```
kappa = ebsd.curvature
```

```
kappa = curvatureTensor (show methods, plot)
  size: 51 x 101
  unit: 1/um
  rank: 2 (3 x 3)
```

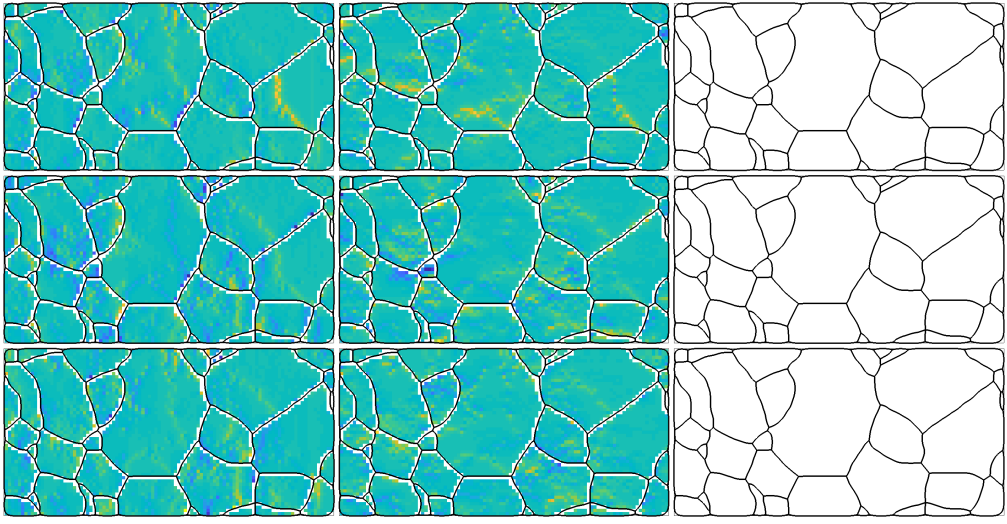
```
kappa(2,3)
```

```
ans = curvatureTensor (show methods, plot)
  unit: 1/um
  rank: 2 (3 x 3)

 *10^-5
  -7.262   12.678      NaN
 -18.454   27.836      NaN
 -20.467   25.478      NaN
```

Geometrically Necessary Dislocations

```
plot(ebsd,kappa{1,1})
```



Geometrically Necessary Dislocations

```
ebsd = loadEBSD('DC06_2undef.ang')
kappa = ebsd.curvature
alpha = kappa.dislocationDensity
```

```
alpha = dislocationDensityTensor (show methods, plot)
  size: 51 x 101
  unit: 1/um
  rank: 2 (3 x 3)
```

```
alpha(2,3)
```

```
ans = dislocationDensityTensor (show methods, plot)
  unit: 1/um
  rank: 2 (3 x 3)

 *10^-5
     NaN -18.454 -20.467
  12.678     NaN  25.478
     NaN     NaN -20.574
```

Geometrically Necessary Dislocations

```
ebsd = loadEBSD('DC06_2undef.ang')
kappa = ebsd.curvature
alpha = kappa.dislocationDensity
dS = dislocationSystem.bcc(ebsd.CS)
```

```
dS = dislocationSystem (show methods, plot)
 mineral: Iron (Alpha) (432)

 edge dislocations : 48 x 1
 Burgers vector  line vector  energy
     [ 1 -1  1]    [-2 -1  1]       2
     [-1  1  1]    [ 0  1 -1]       2
     [-1  1  1]    [-1  4 -5]       2

 screw dislocations: 4 x 1
 Burgers vector  energy
     [-1 -1 -1]       1
     [ 1 -1  1]       1
     [-1  1  1]       1
     [ 1  1 -1]       1
```

Geometrically Necessary Dislocations

```
ebsd = loadEBSD('DC06_2undef.ang')
kappa = ebsd.curvature
alpha = kappa.dislocationDensity
dS = dislocationSystem.bcc(ebsd.CS)
dS(1).tensor
```

```
ans = dislocationDensityTensor (show methods, plot)
  unit   : au
  rank   : 2 (3 x 3)
  mineral: Iron (Alpha) (432)

 -1.1717 -0.5858  0.5858
  1.1717  0.5858 -0.5858
 -1.1717 -0.5858  0.5858
```

Geometrically Necessary Dislocations

```
ebsd = loadEBSD('DC06_2undef.ang')
kappa = ebsd.curvature
alpha = kappa.dislocationDensity
dS = dislocationSystem.bcc(ebsd.CS)
dSSample = ebsd.orientations * dS
```

```
dSSample = dislocationSystem (show methods, plot)
 edge dislocations : 5123 x 48
 screw dislocations: 5123 x 4
```

Geometrically Necessary Dislocations

```
ebsd = loadEBSD('DC06_2undef.ang')
kappa = ebsd.curvature
alpha = kappa.dislocationDensity
dS = dislocationSystem.bcc(ebsd.CS)
dSSample = ebsd.orientations * dS
[rho,factor] = kappa.fitDislocationSystems(dSSample);
```

Geometrically Necessary Dislocations

```
ebsd = loadEBSD('DC06_2undef.ang')
kappa = ebsd.curvature
alpha = kappa.dislocationDensity
dS = dislocationSystem.bcc(ebsd.CS)
dSSample = ebsd.orientations * dS
[rho,factor] = kappa.fitDislocationSystems(dSSample);
alpha = sum(dSSample.tensor .* rho,2)
```

```
alpha(2,3)
```

```
ans = dislocationDensityTensor (show methods, plot)
  unit: 1/um
  rank: 2 (3 x 3)

 *10^-5
 -28.855 -18.454 -20.467
  12.678   6.243  25.478
  -1.337  -1.989 -20.574
```

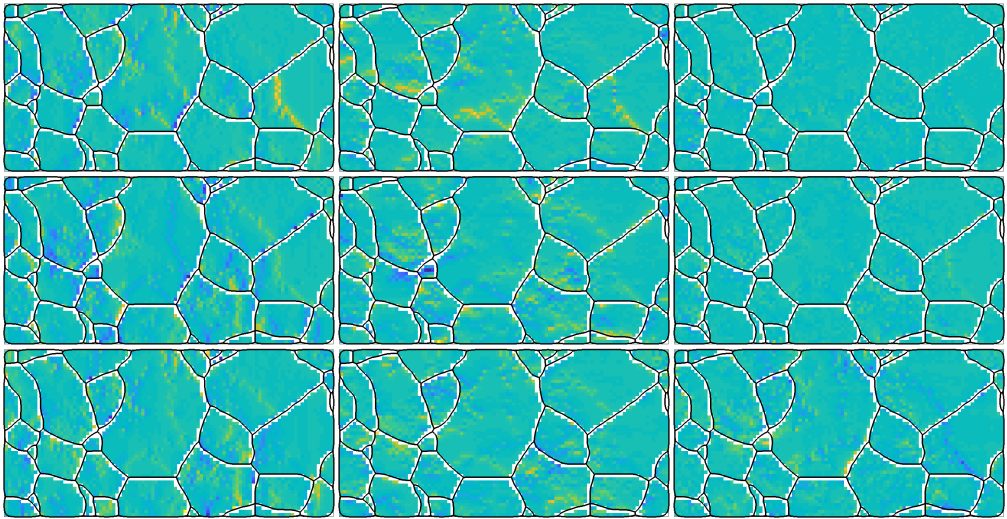
Geometrically Necessary Dislocations

```
ebsd = loadEBSD('DC06_2undef.ang')
kappa = ebsd.curvature
alpha = kappa.dislocationDensity
dS = dislocationSystem.bcc(ebsd.CS)
dSSample = ebsd.orientations * dS
[rho,factor] = kappa.fitDislocationSystems(dSSample);
alpha = sum(dSSample.tensor .* rho,2)
kappa = alpha.curvature
```

```
kappa = curvatureTensor (show methods, plot)
  size: 51 x 101
  unit: 1/um
  rank: 2 (3 x 3)
```

Geometrically Necessary Dislocations

```
plot(ebsd, kappa{1,1})
```



Geometrically Necessary Dislocations

```
plot(ebsd,factor * sum(abs(rho .* dSSample.u),2))
```