OdfPf User Manual—Draft Version

Donald E. Boyce

May 19, 2004

ISBN (to be obtained)

Paul Dawson
196 Rhodes Hall
Cornell University
Ithaca, NY 14853

Contents

Preface

The intent of the manual is to document the polycrystal analysis software developed by the Deformation Process Simulation Laboratory, research group in Mechanical Engineering at Cornell University. The software targets texture analysis—the study of preferred orientation in polycrystal materials. The DPLab group has been using an approach based on finite element methods and Rodrigues parameters for many years, beginning with the doctorate work of Ashish Kumar. We believe this approach has many advantages and we hope other researchers will adopt these methods.

Chapter 1

Overview

This chapter gives an overview of the functionality of the library. The first section discusses the target application—texture analysis. The second section lays out the underlying mathematical model, and the third section is a discussion of the implementation.

1.1 Application

The application of this software is to the study of preferred orientation, or texture, in polycrystalline materials. Such materials are made up of many, often millions, of individual crystals. The properties of the constituent crystals are anisotropic and depend on how the crystal is oriented in space. The aggregate properties of the material therefore depend on the underlying distribution of single crystal orientations. The study of these distributions is the field of texture analysis.

As an example, consider the Young's modulus of a polycrystal under applied extension in a fixed spatial direction. The Young's modulus is the ratio of the stress required to extend the body to the relative extension. For an isotropic material, the ratio is independent of the direction of extension. Crystals are inherently anisotropic, and for crystals the Young's modulus is directional; certain directions are more easily extensible than others. For a given material, the directional Young's modulus depends on how the direction of extension is related to the structural lattice.

One way to model the extension of a polycrystal is to assume that all the constituent crystals undergo the same extension and to compute the macroscopic stress as the average stress of the constituent crystals. In that model, the stresses of the individual crystals depend on how each crystal lattice is oriented relative to the extension. To compute the average stress, one needs to know the volume fraction of crystals at each possible orientation. If the crystals are oriented uniformly, with no particularly preferred orientations, the constitutive model will prove to be isotropic. If certain orientations are more common than others, then the model will be anisotropic and the properties will be closer to those of the preferred orientations. The stress in this model is seen to be an average over the orientation distribution.

The model described above is simplistic in that it takes no account of the spatial distribution crystals and their consequent interactions. Nevertheless, virtually any model of polycrystal behavior accounts for orientation distribution. The crystal orientation is typically modeled as an orthonormal basis, a set of three vectors describing crystallographically distinguishable directions. The model for the distribution is called an *orientation distribution function*, or ODF, and is essentially a density function for the volume fraction of material within any range of orientations.

Experimentally, there are two classes of methods used to study orientation distributions. The first class is scanning methods in which a section of the specimen is scanned and complete orientation data are determined at each point, resulting in map of orientations. An example of this method is EBSD (Electron Backscattered Diffraction) in which an intense beam of electrons is applied to small section of the sample; the pattern produced by the scattered electrons is then examined to determine the crystal orientation at each location. The other class of methods provide pole density measurements. In this class, a beam is applied to a large section of the specimen and diffraction measurements provide information about the strength of reflections off of certain crystallographic planes. The resulting data is in form of *pole figures*, which are diffraction intensity maps on the sphere. These methods do not provide complete orientation data because the measurement only distinguishes one direction of the crystal—the normal to the crystallographic plane under examination. So two different types of data are produced by experimental measurements of texture, and data analysis software needs to accommodate both types.

The orientation distribution is three-dimensional, being based on rotation matrices. There are two common ways to visualize an ODF. The first is by displaying sections of a three-dimensional parameter space. Plots through a space of Euler angle parameters are the most commonly used. The space of Rodrigues parameters is another choice; it's main advantage is the simplicity in which crystal symmetries are expressed. It is the choice of this software package. Orientation distributions are also commonly shown using pole figures, which are integral projections of the orientation distribution onto the sphere. Each crystal-relative direction has a corresponding pole figure. Points on the sphere are interpreted as spatial directions, and the value at a point indicates the density of crystals with the given crystal-relative direction aligned with the spatial direction. Alternatively, one could fix a particular spatial direction and plot the density on the sphere, where points on the sphere are now interpreted as crystal-relative directions. Such a density is called an *inverse pole figure*.

The study of orientation distributions is further complicated by consideration of symmetries. Crystal symmetry refers to rotations which preserve the crystal's structural lattice; two directions related by crystal symmetry are indistinguishable crystallographically. Sample symmetry refers to spatial symmetry of the the distribution. Usually, sample symmetry in an ODF arises because the material's processing history had spatial symmetries. Crystal and sample symmetries are manifested in an ODF through a kind of periodicity.

This library of software is intended to be a set of tools for texture analysis. It is not exhaustive but does provide a set of high-level components which provide a base for further development. The basic treatment is to parameterize orientation space with Rodrigues parameters and to discretize the parameter space with finite elements. Similarly, meshes on the sphere provide the discretization of pole figure data. The thrust of the library is to compute the relation between orientation distributions and pole figures, accounting for crystal symmetry.

1.2 Mathematical Model

Here, we offer our own mathematical formalism for orientations, symmetries and functions of orientations. It is not essentially different than most other presentations of texture analysis, but it is designed to make clear certain distinctions. First, we want to distinguish vectors and tensors from their component matrices. Also we want to differentiate between absolute orientations and orientation classes arising from crystal symmetry. We also lower the importance of the sample reference frame. Many discussions treat an orientation as a tensor taking the sample reference frame to the crystal reference frame, or vice versa. Here, the sample reference frame is merely a source of components for the various vectors and tensors.

The *material space* is a three-dimensional real vector space. It models the space containing a polycrsytal and provides the framework for comparing orientations. An *orientation*, $\mathcal{C}$, is a proper orthonormal basis $(\vec{c}_1, \vec{c}_2, \vec{c}_3)$. It represents a crystal-relative reference frame; each direction has some crystallographically identifiable property. The *sample frame*, $\mathcal{S}$, is a fixed proper orthonormal basis $(\vec{s}_1, \vec{s}_2, \vec{s}_3)$. Its purpose is to provide components for orientations and associated vectors and linear transformations. There is often a natural sample frame related to the material's processing, such as the RD-TD-ND frame used in rolling. The *orientation matrix* of an orientation $\mathcal{C}$ with respect to $\mathcal{S}$ is the matrix C whose i'th column contains the components of $\vec{c}_i$ in the basis $\mathcal{S}$. The orientation matrix is the change of basis matrix taking components in $\mathcal{C}$ to components in $\mathcal{S}$.

A *crystal symmetry group*, $\mathcal{G}$, is a finite subgroup of $SO(3)$. The symmetry group is to be interpreted as the group of crystal-relative transformations which preserve the crystal's structural lattice. Members of the symmetry group act on crystal components and return crystal components. If C is an orientation and $G \in \mathcal{G}$ is a crystal symmetry, thenCG is the symmetrical orientation.

For a particular symmetry group $\mathcal{G}$, an *orientation class* is an equivalence class of orientations in which two orientations are equivalent if one can be derived from the other by application of a symmetry transformation. In terms of orientation matrices, two are equivalent if $C_1 = C_2 G$ for some $G \in \mathcal{G}$. The *orientation space*, $\mathcal{R}$, associated with $\mathcal{G}$ is the quotient manifold of $SO(3)$ defined by the equivalence relation above. A *fundamental region* of $SO(3)$ is an open subset containing at most one representative of each equivalence class, the closure of which contains representatives from all equivalence classes. Fundamental regions can be used to parameterize orientation space.

An *orientation function* f, is a real-valued function on orientation space. It can be associated with a function f^* on all of $SO(3)$ with a periodicity based on the symmetry group, *i.e.* $f^*(RG) = f^*(R)$ for all $G \in \mathcal{G}$. We call such a function $\mathcal{G}$-periodic. A special kind of orientation function is an *orientation distribution function*, or ODF, which has the property that is everywhere positive and has unit integral. ODF's are commonly displayed in multiples of uniform density (MUD); in that case the ODF is rescaled to have integral mean value of one.

Functions on the sphere, S^2, are also of interest. The analogue of the ODF on the sphere is the *pole density function*, or PDF, also known as a pole figure. It also has the properties of positivity and unit integral. Likewise pole figures are commonly displayed multiples of uniform.

Orientations associate naturally with crystal and sample directions, *i.e.* points on the sphere. If c is a fixed set of (crystal) components, then each orientation $\mathcal{C}$ can be associated with a spatial direction by by taking a linear combination of the the crystal basis vectors with coefficients given by c. In terms of matrices, the orientation matrix R maps to the (spatial) component vector Rc. Likewise if $\vec{s}$ is a fixed spatial direction, then an orientaion $\mathcal{C}$ can be associated with the (crystal) components of $\vec{s}$. The matrix form of the map takes a matrix R to $R^T s$, where s is the (sample) component vector of $\vec{(s)}$.

For a fixed spatial direction $\vec{s}$ and a given crystal direction c, there are many orientations for which c aligns with $\vec{s}$. Such a set of crystal directions is the *fiber* associated with c and $\vec{s}$. The fiber is in fact a geodesic in $SO(3)$. In terms of orientation matrices, the fiber is the set of rotations R such that $Rc = s$, where s is the vector of sample components of $\vec{s}$.

If f is a (smooth) function of orientations and $\mathcal{F}$ is the fiber associated with c and $\vec{s}$, then one can integrate f over the fiber to obtain the *fiber integral*. Using matrices, a convenient notation for the fiber integral is:

$$P(c, s) \;=\; \int_{Rc=s} f dR.$$

We shall refer to P as the *fiber integral function*. If we fix the crystal direction c, then $P_c(s) = P(c, \cdot)$ is the c-pole figure associated f. Likewise, if we fix s, then $P_s(c) = P(\cdot, s)$ is the s-inverse pole figure associated f. The operators which take f to P_c and P_s respectively are linear. One further property is important. If the function f is $\mathcal{G}$-periodic, then $P(Gc, s) = P(c, s)$ for all $G \in \mathcal{G}$. The utility of this result is that in order to evaluate pole figures for orientation classes, the fiber integral needs only to be computed for one fiber associated with any symmetrically equivalent crystal direction.

1.3 Implementation

1.3.1 Representation of Orientations

Orientation matrices can represented in numerous ways. The basis for this package is the *unit quaternion representation.* A unit quaternion is a four dimensional vector of unit length with a certain group operation defined. There is a two-to-one correspondence of unit quaternions with rotation matrices in $SO(3)$—each quaternion and its negative refer to the same rotation. Locally, this correspondence is an isometry (up to a constant scaling). Furthermore, there is an algebraic correspondence (homomorphism) of the quaternion group operation with the multiplication of rotation matrices. The quaternion representation provides a lower-dimensional and simpler computational framework for manipulating orientations.

There are various three-dimensional parameterizations of orientations. The traditional descriptions of orientation have used *Euler angles*, in which a rotation is described by three consecutive rotations about certain axes. The choice of a particular set of axes leads to conventions commonly used in the field. The choice of axes $(\vec{s}_3, \vec{s}_1, \vec{s}_3)$ gives the Bunge convention, whereas the choice $(\vec{s}_2, \vec{s}_1, \vec{s}_2)$ leads to the Roe convention. Many other conventions exist.

A number of other representations are based on the exponential map and involve parameterization by the axis of rotation scaled by some function of the rotation angle. Every rotation is the exponential of some skew matrix (in fact, of infinitely many skew matrices). If W is a skew matrix, then it has an *axial vector* with the property that $Wx = w \times x$ for all x. The axial vector is an eigenvector of W associated with the zero eigenvalue and consequently is an eigenvector of $\exp W$ associated with the eigenvalue one. Thus the axial vector of W is the axis of rotation for $\exp W$. It can also be shown that the angle of rotation is the magnitude of w. This provides the most basic of the angle/axis parameterizations; the parameter is a 3-vector which maps to a rotation about that vector through an angle of the vector's magnitude.

More angle/axis parameterizations can be obtained by radial rescalings of the axial vector. A particularly interesting choice is the *Rodrigues parameterization* in which the parameter is scaled so that it has length $\tan \theta/2$, where θ is the angle of rotation. Any rotation through an angle less than by π has a unique Rodrigues parameter associated with it. Rodrigues parameters are undefined for binary rotations *i.e.* rotations with angle π.

Because of certain properties, the Rodrigues parameterization is ideally suited for working with symmetries. Using Rodrigues parameters, one can always parameterize a fundamental region with a polyhedral domain. The symmetry group operations always reduce the parameter by intersection with a half-space. Consequently, each symmetry group is associated with a particular polyhedron. Another useful property is that fibers are parameterized by lines in Rodrigues parameter space.

1.3.2 Discretization

The numerical approach to modeling the ODF's and pole figures is based on finite element methods. Parameter spaces for both the fundamental region of orientation space and the sphere are discretized using linear, simplicial elements (triangles and tetrahedra). In the case of orientations with symmetry, we form a mesh over the Rodrigues parameters for a fundamental region of $SO(3)$. For spheres, we create a mesh by inscribing triangles and interpret each element as a parameter space for the map to the sphere defined by radial projection. This discretization also works on S^3 with inscribed tetrahedra and can be used as a method for modeling orientations with or without crystal symmetries. All of the meshes described above are on parameter spaces and involve subsequent mappings.

For Rodrigues meshes, the boundary is of particular interest. Because of the crystal symmetries, two or more points on the boundary may refer to equivalent orientations. The actual equivalence depends on the crystal symmetry associated with the particular boundary face. If the face is associated with a symmetry through angle ϕ about axis n, then each point on the face is equivalent by symmetry to a point on the opposite face rotated again about the axis n but through an angle of $\phi/2$. In order to have continuous approximation on the mesh, the surface elements need to respect the symmetries. Obeying the boundary equivalences adds complexity to the meshing process.

Given meshes for the orientation space and for the sphere, one can compute the discrete form of the fiber integral function (pole figure projection). For each target point on the sphere, usually the set of mesh nodal points, we first compute points along the fiber. As mentioned above, this could be done extremely accurately using lines through the Rodrigues mesh. However, that approach may be too involved using a high-level programming language such as Matlab® and may incur a big performance hit. Instead, we directly generate a large number of equally spaced points along the quaternion fiber and convert to Rodrigues parameters for function evaluation. Because only a subset of the nodal points on the Rodrigues mesh will contribute to any particular value on the sphere, the result of this process is sparse matrix which takes nodal point values on the Rodrigues mesh to pole figure values at the target points.

Finally, we offer the following discrete formulation for the pole figure inversion problem, illustrated by example in 2.4. Because the pole figure projection takes a three-dimensional distribution to a two-dimensional one, much information is lost and is not recoverable. Furthermore, when more than one pole figure is given, consistency conditions need to be met in order for an underlying ODF to exist. We offer a two-step approach. First, we find the best consistent pole figure data by finding an ODF which produces pole figures as near the data as possible. Then, we find the ODF which is minimum in gradient norm while matching the best consistent pole figures to within a specified tolerance. This is a very general approach and takes advantage the high-level optimization routines in Matlab®. The approach can be adopted in virtually any situation, including full pole figure data, incomplete pole figure data, inverse pole figure data or just scattered pole figure values.

The pole figure inversion problem is interesting mathematically because the high degree of indeterminacy of the continuum problem is not necessarily manifested in the discrete problem. For the finite element discretization in particular, pole figure matrices are often of full rank. Nevertheless, that does not mean that the second optimization step is not necessary. Because of the massive indeterminacy of the continuum problem, some control over the ODF is necessary in order to obtain a result which is consistent with mesh refinement (convergence).

Chapter 2

Examples

This section gives illustrative problems that can be solved using the library.

2.1 Building a Workspace

The first example illustrates how to get started. It builds the basic mesh structures for the sphere and for the orientation space, as well as supplying further useful information about them. It then constructs matrices for producing pole figures or inverse pole figures.

There are three scripts, `CubicWS`, `HexagonalWS` and `OrthorhombicWS`, all of which run the same basic function `Workspace` with different parameters. The bulk of the work is done in `Workspace`. It accepts base meshes for the orientation space and for the sphere, and it is driven by a series of keyword arguments. It returns a structure containing the FR mesh, one for the sphere mesh and another structure which holds the hkl's and matrices for any specified pole figures. The mesh structures have additional components containing additional information about the meshes.

Workspace.m

```
function ws = Workspace(frmesh, sphmesh, varargin)
% Workspace - Make Odf/Pf workspace.
%
%   USAGE:
%
```

```
%   ws = Workspace(frmesh, sphmesh, 'param', 'value')
%
%   INPUT:
%
%   frmesh  is a MeshStructure,
%           on a fundamental region; it is expected to
%           have an additional structure component, 'symmetries'
%   sphmesh is a MeshStructure,
%           on the sphere
%
%   OUTPUT:
%
%   ws is a structure
%
ws.frmesh  = frmesh;
ws.frmesh.numind = size(frmesh.crd, 2) - size(frmesh.eqv, 2);
%
ws.sphmesh = sphmesh;
%
%--------------------Defaults and Options------------------------------
%
optcell = {...
    'PointsPerFiber',   100,  ...
    'MakePoleFigures', {{}},  ...
    'MakeFRL2IP',     'off',  ...
    'MakeFRH1IP',     'off',  ...
    'MakeSphL2IP',    'off',  ...
    'MakeSphH1IP',    'off',  ...
    'ProgressReport',  'on',  ...
    'QRuleTetrahedra', 'qr_tetd06p24',  ...
    'QRuleTriangles',  'qr_trid06p12',  ...
    'NoOption',          []
    };
%
opts = OptArgs(optcell, varargin);
%
report = OnOrOff(opts.ProgressReport);
%
```

```
%-------------------- Execution
%
%  Make requested pole figures.
%
pf_hkls   = opts.MakePoleFigures;
if (~isempty(pf_hkls))
  ws.pfmats = struct('hkl', pf_hkls);
end
%
for i=1:length(pf_hkls)
  %
  hkl   = pf_hkls{i};
  ppf   = opts.PointsPerFiber;
  block = max(floor(5.0e4/ppf), 1); % 50,000 points at once
  %
  if report
    disp(['working on OdfPf matrix:  ', num2str(i)]);
    disp(['   hkl:  ', num2str(hkl(:))']);
  end
  %
  ws.pfmats(i).odfpf = ...
      BuildOdfPfMatrix(hkl, ...
       frmesh, frmesh.symmetries, ...
       sphmesh.crd, ppf, 0, block);
  %
end
%
%  Make various quadratic forms.
%
%  *** Fundamental Region
%
makefrl2 = OnOrOff(opts.MakeFRL2IP);
makefrh1 = OnOrOff(opts.MakeFRH1IP);
%
if (makefrl2 | makefrh1)
  qrule3d = LoadQuadrature(opts.QRuleTetrahedra);
  gqrule  = QRuleGlobal(frmesh, qrule3d, @RodMetric);
  npqp    = NpQpMatrix(frmesh, qrule3d);
  ws.frmesh.qrule = qrule3d;
end
%
if (makefrl2)
  if report
    disp(['working on FR L2IP matrix:  ']);
  end
  ws.frmesh.l2ip = L2IPMatrix(gqrule, npqp);
end
%
if (makefrh1)
  if report
    disp(['working on FR H1IP matrix:  ']);
  end
  ws.frmesh.h1form = H1SIPMatrix(frmesh, qrule3d, ...
 RodDifferential(frmesh, qrule3d.pts));
end
%
%  *** Sphere
%
makesphl2 = OnOrOff(opts.MakeSphL2IP);
makesphh1 = OnOrOff(opts.MakeSphH1IP);
%
if (makesphl2 | makesphh1)
  qrule2d = LoadQuadrature(opts.QRuleTriangles);
  gqrule  = QRuleGlobal(sphmesh, qrule2d, @RodMetric);
  npqp    = NpQpMatrix(sphmesh, qrule2d);
  ws.sphmesh.qrule = qrule2d;
end
%
if (makesphl2)
  if report
    disp(['working on Sphere L2IP matrix:  ']);
  end
  [gqr, ws.sphmesh.l2ip] = SphGQRule(sphmesh, qrule2d);
end
%
if (makesphh1)
```

```
  if report
    disp(['working on Sphere H1SIP matrix:  ']);
  end
  ws.sphmesh.h1form = SphH1SIP(sphmesh, qrule2d);
end
%
return
```

CubicWS.m

```
%
%  Generate workspaces for cubic fundamental region.
%
%-------------------- User Input
%
fr_refine =   3;  % refinement level on FR
sp_refine =  10;  % refinement level on sphere
per_fiber = 100;  % points per fiber
%
pf_hkls ={[1 1 1], [1 0 0], [1 1 0] };
%
wsopts = {...
    'MakePoleFigures',   pf_hkls, ...
    'PointsPerFiber',  per_fiber, ...
    'MakeFRL2IP',           'on', ...
    'MakeFRH1IP',           'on', ...
    'MakeSphL2IP',          'on', ...
    'MakeSphH1IP',          'on'  ...
 };
%
%-------------------- Build workspace
%
cfr0 = CubBaseMesh;
csym = CubSymmetries;
cfr  = RefineMesh(cfr0, fr_refine, csym);
cfr.symmetries = csym;
%
sph0 = SphBaseMesh(2, 'Hemisphere', 'on'); % 2d sphere
```

```
sph  = RefineMesh(sph0, sp_refine);
sph.crd = UnitVector(sph.crd);
%
wscub = Workspace(cfr, sph, wsopts{:});
save wscub wscub wsopts
```

2.2 Making Pole Figures

This example illustrates the construction of pole figures from an ODF. The process of making pole figures from an ODF is just matrix multiplication, so the highlighted routines are the interactive graphics. Orientation functions are displayed using `PlotFR`, and functions on the sphere are shown using `PlotSphere`.

MakePoleFigures.m

```
%
%  Make pole figures.
%
%-------------------- User Input
%
bground = 0.5;  % background (% of uniform)
centers = [
    0.25    -0.2
    0.25     0.1
    0.25     0.0
    ];
%
stdevs  = [0.3, 0.2];     % standard deviations
%
%-------------------- Execution
%
%  Load a workspace.
%
addpath('../build-workspaces'); % use \ for windows
load wscub
%
```

```
%  Create sample ODF.
%
allpts = wscub.frmesh.crd;
nindep = size(allpts, 2) - size(wscub.frmesh.eqv, 2);
pts    = allpts(:, 1:nindep);  % independent nodes
%
symm  = CubSymmetries;
odf   = zeros(1, nindep);
cenfr = ToFundamentalRegion(QuatOfRod(centers), symm);
%
for i=1:size(centers, 2)
  rg  = RodGaussian(cenfr(:, i), pts, stdevs(i), symm);
  odf = odf + rg ./ MeanValue(rg, wscub.frmesh.l2ip);
end
%
odf  = odf ./ MeanValue(odf, wscub.frmesh.l2ip);
%
unif = ones(1, nindep);
unif = unif ./ MeanValue(unif, wscub.frmesh.l2ip);
%
odf = bground*unif + (1-bground)*odf;
%
%  Plot the odf.
%
fropts = {'Symmetries', 'cubic', 'ShowMesh', 'on'};
PlotFR(wscub.frmesh, odf, fropts{:});
%
%  Make and display the pole figure.
%
pf111 = wscub.pfmats(1).odfpf*odf(:);
%
plotopts = {'UpperHemisphere', 'on'};
PlotSphere(wscub.sphmesh, pf111, plotopts{:});
%
pf100 = wscub.pfmats(2).odfpf*odf(:);
%
PlotSphere(wscub.sphmesh, pf100, plotopts{:});
%
```

```
pfdata = {pf111, eye(3), pf100, eye(3)};
%
PlotPF2d(wscub.sphmesh, pfdata);
%
%  Create DX output files.
%
Ndata = {'odf', odf};
ExportDX('cubodf', wscub.frmesh, Ndata);
%
Ndata = {'pf111', pf111, 'pf100', pf100};
ExportDX('cubpfs', wscub.sphmesh, Ndata);
%
save
```

2.3 ODF From Aggregate

Data from simulations and experiements often come in the form of a discrete collection of orientations with or without weights. Here we illustrate how to construct an ODF from complete orientation data.

The fastest way is to use the `DiscreteDelta` function. It treats the data points as point sources acting on the mesh function space. It's advantage is that it is fast; it's problem is that it can (and usually does) produce negative values. The result could be smoothed with `SmoothFunction`, which performs a convolution with a local averaging function, however that takes too long to be practical even for moderate sized meshes. The other option is `AggregateFunction` which simply associates each data point with a standard distribution (such as a Gaussian) centered at that point and then accumulates the result into a final distribution. This is more intensive than `DiscreteDelta`, but it never produces negative values.

OdfFromAggregate.m

```
%
%  ODF From Aggregate.
%
%  This script illustrates the methods available for
%  constructing an ODF from a collection of individual
%  points.
%
%
%-------------------- User Input
%
wsname = 'wscub';      % workspace name
numpts = 2000;         % number of points
std_agg = deg2rad(5);  % std deviation for AggregateFunction
std_sm  = deg2rad(10); % std deviation for smoothing
%
%-------------------- Execution
%
%  Load workspace for fundamental region.
%
addpath('../build-workspaces');
load(wsname);
eval(['ws = ', wsname, ';']);
clear(wsname);
%
%  Generate a random aggregate.
%
wts  = ones(1, numpts);
quat = UnitVector(randn(4, numpts));
rod  = ToFundamentalRegion(quat, ws.frmesh.symmetries);
%
%  Method 1:  DiscreteDelta
%
tic
[elem, ecrd] = MeshCoordinates(ws.frmesh, rod);
odf1 = DiscreteDelta(ws.frmesh, ws.frmesh.l2ip, elem, ecrd, wts);
odf1 = odf1./MeanValue(odf1, ws.frmesh.l2ip);
t = toc;
disp(['Time for DiscreteDelta:  ', num2str(t)]);
%
%  Method 1a:  Smoothed DiscreteDelta
%
gqrule = QRuleGlobal(ws.frmesh, ws.frmesh.qrule, @RodMetric);
tic
```

```
odf1a = SmoothFunction(odf1, ws.frmesh, ...
       ws.frmesh.qrule.pts, gqrule, ...
       @RodGaussian, std_sm, ws.frmesh.symmetries);
odf1a = odf1a ./MeanValue(odf1a, ws.frmesh.l2ip);
t = toc;
disp(['Time for SmoothFunction:  ', num2str(t)]);
%
%  Method 2:  AggregateFunction
%
pts   = ws.frmesh.crd(:, 1:ws.frmesh.numind);
agg   = rod;
wts   = ones(1, numpts);
%
PointFun = @RodGaussian;
stdev    = deg2rad(5);
sym      = CubSymmetries;
%
tic
odf2 = AggregateFunction(pts, agg, wts, PointFun, stdev, sym);
odf2 = odf2./MeanValue(odf2, ws.frmesh.l2ip);
t = toc;
disp(['Time for AggregateFunction:  ', num2str(t)]);
%
save odfs
%
%  Create DX output files.
%
Ndata = {'odf1', odf1', 'odf1a', odf1a', 'odf2', odf2'};
ExportDX('agg-odf', ws.frmesh, Ndata);
```

2.4 ODF From Pole Figures

Often, experimental data come in the form of pole figures. It is desirable to find an ODF consistent with the pole figure data as much as possible. Since pole figures are two-dimensional projections of a three-dimensional distribution, there is a high degree of indeterminacy in the process. The example here illustrates one possible approach based on two consecutive optimizations. The first optimization is to find the the pole figures which match the original ones as nearly as possible while being consistent in the sense that they can be derived from a single ODF. The second optimization involves controlling the ODF while still matching the best pole figures. There are no particular routines from this package which are highlighted. Instead, the optimization toolbox of Matlab® is the major player.

OdfFromPfs.m

```
%
%  ODF From Pole Figures.
%
%  This script illustrates the methods available for
%  constructing an ODF from pole figure data.
%
%
%-------------------- User Input
%
wsname = 'wspfi';      % workspace name
h1tol  = 1.0e-1;
%
%-------------------- Execution
%
%  Load workspace for fundamental region.
%
addpath('../build-workspaces');
%
load(wsname);
eval(['ws = ', wsname, ';']);
clear(wsname);
%
% 1) Find best pole figure.
%
% X=QUADPROG(H,f,A,b,Aeq,beq,LB,UB,X0)
%
sphl2 = ws.sphmesh.l2ip;
%
H = ws.pfmats(1).odfpf'*sphl2*ws.pfmats(1).odfpf + ...
```

```
    ws.pfmats(2).odfpf'*sphl2*ws.pfmats(2).odfpf;
%
f = -1*(...
    ws.pfmats(1).odfpf'*sphl2*ws.pfs(1).data + ...
    ws.pfmats(2).odfpf'*sphl2*ws.pfs(2).data  ...
    );
%
A   = [];
b   = [];
Aeq = full(sum(ws.frmesh.l2ip));
beq = sum(Aeq);
LB  = zeros(1, ws.frmesh.numind);
UB  = [];
X0  = ones(ws.frmesh.numind, 1);
%
disp('running first optimization');
odf1 = quadprog(H,f,A,b,Aeq,beq,LB,UB,X0);
bestpf1 = ws.pfmats(1).odfpf * odf1;
bestpf2 = ws.pfmats(2).odfpf * odf1;
%
% 1) Find best ODF.
%
% X=QUADPROG(H,f,A,b,Aeq,beq,LB,UB,X0)
%
H = ws.frmesh.h1form;
f = zeros(ws.frmesh.numind, 1);
%
A1 = [...
    ws.pfmats(1).odfpf; ...
    ws.pfmats(2).odfpf  ...
    ];
b1  = A1*odf1;
tol = h1tol*ones(size(b1));
%
A  = [A1; -A1];
b  = [b1 + tol; -b1 + tol];
%
%  Aeq and beq as before.
```

```
%
X0 = odf1;
%
disp('running second optimization');
odf = quadprog(H,f,A,b,Aeq,beq,LB,UB,X0);
finalpf1 = ws.pfmats(1).odfpf * odf;
finalpf2 = ws.pfmats(2).odfpf * odf;
%
%  Create DX output files.
%
Ndata = {'odf1', odf1, 'odf', odf};
ExportDX('pfs-odf', ws.frmesh, Ndata);
%
Ndata = {'raw-110', ws.pfs(1).data, 'raw-100', ws.pfs(2).data, ...
'best-110', bestpf1, 'best-100', bestpf2, ...
'final-110', finalpf1, 'final-100', finalpf2};
%
save
```

2.5 Average Properties

This is a short example illustrating how to integrate a function over the mesh. The Taylor factor data come from a polycrystal plasticity simulation of uniaxial tension of a cubic material. The data correspond to nodal point values for the given mesh. Integrals are calculated using L^2 inner product matrix.

AverageProperties.m

```
%
%  Average properties.
%
mymesh  = LoadMesh('fr-cubic-2');
myqrule = LoadQuadrature('qr_tetd08p43');
gqr     = QRuleGlobal(mymesh, myqrule, @RodMetric);
l2ip    = L2IPMatrix(gqr, NpQpMatrix(mymesh, myqrule));
%
```

```
tayfac = load('taylor-factors.dat');
%
wts = sum(l2ip);
avgtayfac = (wts*tayfac)/sum(wts)
```

Chapter 3

Functions

This chapter gives the interactive help for the functions in the library.

3.1 Export

ExportDX

```
ExportDX - Write a DX field in DX native format.

USAGE:

header = ExportDX(fname, mesh)
header = ExportDX(fname, mesh, Ndata)
header = ExportDX(fname, mesh, Ndata, Edata)

INPUT:

fname is a string,
      the basename of the DX header file, as well as the basename of the
      various ASCII output files, and also the name of the resulting DX field

mesh  is a MeshStructure

Ndata is a cell array,
      it contains position-dependent data; it has the form {name1, array1,
      name2, array2, ...}, where the names are strings with no spaces and the
      arrays are real-valued data arrays; Ndata can be empty.

Edata is a cell array,
      it contains connection-dependent data and has the same form as Ndata

OUTPUT:

header is a cell array of strings,
       it contains the DX header file content
```

```
NOTES:

*  ExportDX writes the DX field to fname.dx, and the field
   object is named fname. Each array in Ndata or Edata is written to a separate
   file (fname-arayname.dat) and is added as a component of the DX field.

*  Currently, this routine can only handle element types
   of 'lines', 'triangles', 'tetrahedra' or 'cubes'.
```

WriteDataFile

```
WriteDataFile - Write data to a file with a text header.

USAGE:

WriteDataFile(fname, text, data)
WriteDataFile(fname, text, data, basefmt)
WriteDataFile(fname, text, data, basefmt, append)

INPUT:

fname is a string,
      the name of the file
text  is a cell array of strings,
      the text header to write at the beginning of the file; it can be empty
data  is m x n, (numeric)
      the data to write; the data is written as passed, m rows of n columns each
basefmt is a string, (optional, default = '%25.16e')
      the printf-style format to use on each value; if this argument is left
      out or is the empty string '', the default value for reals is used; if
      the argument is the string 'integer', the format used is '%10d'; other
      values are used without change
append is a scalar,  (optional, default = 0)
       if nonzero, the output file is appended to instead of overwritten

OUTPUT:  none
```

WriteMPSOrientations

```
WriteMPSOrientations - Write orientations in MPS format.

USAGE:

header = WriteMPSOrientations(filename,orient, hinfo)

INPUT:

filename is a string,
         the name of the output file
orient   is m x n,
         the array of orientations to write
hinfo    is a cell array,
         it contains header information; options are:
```

```
'Weights',          array

'Hardnesses',       array

'Parameterization', string  [required]

'Angles',           string  ('degrees'|'radians')
```

OUTPUT:

```
header is a cell array of strings,
       it contains the MPS (material point simulator) header information; the
       header and data are written to the requested file
```

3.2 FiniteElement

CheckMesh

CheckMesh - Check that array sizes match for mesh structure.

USAGE:

```
CheckMesh(mesh)
CheckMesh(mesh, vector)
```

INPUT:

```
mesh   is a MeshStructure vector is a vector, (optional)
       a reduced vector of values on the mesh nodal points
```

OUTPUT: calls ‘error’ if inconsistency is found

NOTES:

This routine checks that the coordinates and equivalence array of a mesh are consistent. If the vector argument is given, it checks that it is of the correct length. If any array sizes are inconsistent, it prints an error message and returns.

DiscreteDelta

DiscreteDelta - Evaluate sum of discrete delta functions.

USAGE:

```
fun = DiscreteDelta(mesh, l2ip, elem, ecrd, wts)
```

INPUT:

```
mesh is a MeshStructure l2ip is n x n,
     the inner product matrix for the mesh, where n is the number of
     independent degrees of freedom
elem is an n-vector of integers,
     the list of elements containing the delta-function points
ecrd is m x n,
     the list of barycentric coordinates of the delta-function points
wts  is an n-vector,(optional)
     it gives the weights of each point; if not present, the weights are set to
     one

OUTPUT:

fun is an n-vector,
    the nodal point values of the sum of the discrete delta functions
    associated with the points and weights

NOTES:

*  The resulting function is not normalized in any way.
```

EqvReduce

```
EqvReduce - Reduce matrix columnwise by equivalences.

USAGE:

rmat = EqvReduce(mat, eqv)

INPUT:

mat is m x n, (usually sparse) a matrix, the columns of which
              have underlying nodal point equivalences
eqv is 2 x k, the equivalence array

OUTPUT:

rmat is m x (n-k), the new matrix for the reduced (condensed)
              set of nodes formed by adding equivalent columns to the master
              column

NOTES:

*  The columns of the unreduced nodes are added to the
   columns of the master node.

*  This routine only reduces along the column dimension.  To
   reduce along the rows, apply to the transpose. To reduce along both
   dimensions, call it twice.

*  The equivalence array can be empty, in which case nothing
   is done, and the original matrix is returned.
```

EvalMeshFunc

```
EvalMeshFunc - Evaluate mesh function.
```

```
USAGE:

values = EvalMeshFunc(mesh, fun, els, ecrds)

INPUT:

mesh  is a MeshStructure fun   is an n-vector,
      function values on the set of independent nodal points
els   is an m-vector of integers,
      it gives the list of elements containing the points at which to evaluate
      the function
ecrds is k x m,
      it is the list of barycentric coordinates of the points at which to
      evaluate the function

OUTPUT:

values is an n-vector,
       the values of the function at the specified points
```

H1SIPMatrix

```
H1SIPMatrix -- H^1 semi-inner product [Documentation out of date!]

USAGE:

h1sip = H1SIPMatrix(mesh, qrule, diff)

INPUT:

diff is d1 x d2 x n
     the list of tangent vectors at each of n points
npqp is m x n,
     the matrix which takes nodal point values to (global) quadrature points

OUTPUT:

h1sip is n x n, sparse
    it is the matrix of the H^1 semi-inner product,
 where n is the number of independent degrees of freedom associated with the
 mesh
```

Jacobian

```
Jacobian - Compute Jacobian of linear mesh mappings.

USAGE:

jac = Jacobian(mesh)

INPUT:

mesh is a MeshStructure,
     with simplicial element type

OUTPUT:

jac is 1 x m,
    the Jacobian of each element

NOTES:

*  The mesh may be embedded in a space of higher
   dimension than the reference element. In that case, the Jacobian is computed
   as (sqrt(det(J'*J)) and is always positive. When the target space is of the
   same dimension as the reference element, the Jacobian is computed as usual
   and can be positive or negative.

*  Only simplicial (linear) element types are allowed.
```

L2IPMatrix

```
L2IPMatrix - Form matrix for L2 inner product.

USAGE:

l2ip = L2IPMatrix(gqrule, npqp)

INPUT:

qrule is a QRuleStructure,
      it is the global quadrature rule associated with the underlying mesh
npqp  is m x n, (sparse)
      it is the matrix which takes nodal point values to quadrature point values

OUTPUT:

l2ip is n x n, (sparse)
     the inner product matrix associated with the mesh
```

LoadMesh

```
LoadMesh - Load mesh from a ascii file.

USAGE:

mesh = LoadMesh(name)

INPUT:

name is a string,
    the basename of the mesh files

OUTPUT:

mesh is a MeshStructure,
     for the mesh being loaded
```

```
NOTES:

* Expected file suffixes are:

     .crd for nodal point coordinates, .con for connectivity .eqv for
     equivalences (optional)
```

LoadQuadrature

```
LoadQuadrature - Load quadrature data rule.

USAGE:

qrule = LoadQuadrature(qname)

INPUT:

qname is a string,
      the basename of the quadrature data files

OUTPUT:

qrule is a QRuleStructure,
      it consists of the quadrature point locations and weights

NOTES:

*  It is expected that the quadrature rules are for simplices,
   and the last barycentric coordinate is appended to the file read from the
   data.

*  Expected suffixes are .qps for the location and .qpw for
   the weights.
```

MeanValue

```
MeanValue - Find integral mean value of function.

USAGE:

mv = MeanValue(f, l2ip)

INPUT:

f    is an n-vector,
     an array of nodal point function values
l2ip is n x n,
     the (L2) inner product matrix for the underlying mesh

OUTPUT:

mv is 1 x 1,
   the mean value of ‘f’---the integral of f divided by the measure of the
```

```
  region
```

MeshFaces

```
MeshFaces - Find lower dimensional faces for a mesh.

USAGE:

[faces, (opt) mult] = MeshFaces(con)

INPUT:

con is d x n,
    the mesh connectivity (tetrahedra or triangles)

OUTPUT:

faces is (d-1) x m, (integer)
      the connectivity of the lower dimensional
                    faces of con
mult is 1 x m,  (integer, optional)
     the multiplicity of each face; this should be either 1 (for a surface
     face) or 2 (for an interior face)
```

MeshInfo

```
MeshInfo - Verify and print properties of mesh.

USAGE:

outinfo = MeshInfo(mesh)
outinfo = MeshInfo(mesh, sym)
outinfo = MeshInfo(mesh, sym, tol)

INPUT:

mesh is a MeshStructure sym  is 4 x n,  (optional)
     the symmetry group in quaternions; this argument can be omitted if the
     equivalence array is empty or if you do not desire to verify it
tol  is a scalar, (optional, default = 1.0e-7)
     the tolerance used in verifying and generating equivalences

OUTPUT:  none

This function prints various information to the screen.
```

MeshStructure

```
MeshStructure - Create mesh structure from mesh data.

USAGE:

mesh = MeshStructure
mesh = MeshStructure(crd, con)
mesh = MeshStructure(crd, con, eqv)

INPUT:

crd is e x n,
    the array of nodal point locations
con is d x m, (integer)
    the mesh connectivity
eqv is 2 x k, (integer, optional)
    the equivalence array

OUTPUT:

mesh is a MeshStructure,
     the basic MeshStructure consists of three fields, the nodal point
     coordinates (.crd), the connectivity (.con) and the nodal point
     equivalence array (.eqv).

NOTES:

*  With no arguments, this function returns and empty mesh
   structure. With only two arguments, it sets the equivalence array to be
   empty.
```

NpQpMatrix

```
NpQpMatrix - Nodal point values to quadrature points.

USAGE:

npqp = NpQpMatrix(mesh, qrule)

INPUT:

mesh  is a MeshStructure qrule is a QRuleStructure,
      a quadrature rule on the reference element

OUTPUT:

npqp is m x n, (sparse)
     it is the matrix which takes the values at the independent nodes of the
     mesh to the values at the quadrature points; here, n is the number of
     independent nodal values, and m = q*e, where e is the number of elements
     in the mesh and q is the number of qudrature points per element
```

QRuleGlobal

```
QRuleGlobal - Construct a global quadrature rule for a mesh.

USAGE:

gqrule = QRuleGlobal(mesh, qrule)
gqrule = QRuleGlobal(mesh, qrule, @MetricFun)
```

```
INPUT:

mesh  is a MeshStructure qrule is a QRuleStructure,
      the quadrature rule for the reference element

MetricFun is a function handle, (optional)
          It returns the volumetric integration factors for the applied
          mapping. It is for the case in which the mesh is the parameter space
          for some manifold and has a subsequent mapping associated with each
          point.

          met = Metricfun(pts)

          pts is m x n array of points met is 1 x n vector of integration
          factors

OUTPUT:

NOTES:

*  For spheres, this routine is superceded by ‘SphGQRule’,
   which handles the metric dependency on the mesh. This still is to be used
   for meshes on Rodrigues parameters, for which the metric function is
   independent of the mesh, or for meshes with no further mapping involved.
```

QRuleStructure

```
QRuleStructure - Quadrature rule structure.

USAGE:

qrule = QRuleStructure
qrule = QRuleStructure(pts, wts)

INPUT:

pts is m x n,
    a list of n m-dimensional points
wts is an n-vector,
    the associated weights

OUTPUT:

qrule is a QRuleStructure,
      it consists of two fields, points (.pts) and weights (.wts)

NOTES:

*  With no arguments, this returns an empty structure.
```

ReduceMesh

```
ReduceMesh - Find equivalence array for a Rodrigues mesh.
```

```
USAGE:

newmesh = ReduceMesh(mesh, sym)
newmesh = ReduceMesh(mesh, sym, tol)

INPUT:

mesh is a MeshStructure on the fundamental region
     with (possibly) an empty equivalence array
sym  is 4 x m,
     the symmetry group in quaternions
tol  is 1 x 1, (optional, default: 10^-14)
     the tolerance used for comparing equivalent rotations;

OUTPUT:

newmesh is a MeshStructure on the fundamental region;
        it has the same and connectivity, but reordered; the equivalence array
        is created and the nodes are numbered so that all the independent nodes
        precede the dependent nodes.
```

RefDerivatives

```
RefDerivatives - Reference shape function derivatives.

USAGE:

der = RefDerivatives(n)

INPUT:

n is a positive integer,
  the dimension of the simplex

OUTPUT:

der is n x (n+1),
 column j contains the gradient of the j'th
    barycentric coordinate with respect to the coordinate directions
```

RefineMesh

```
RefineMesh - Refine a simplex-based mesh.

USAGE:

rmesh = RefineMesh(mesh, n)
rmesh = RefineMesh(mesh, n, sym)

INPUT:

mesh is a MeshStructure,
     using 1D, 2D, or 3D simplicial elements
```

```
n    is a scalar, (positive integer)
     the subdivision parameter for each simplex
sym  is 4 x s, (optional)
     the array of quaternions giving the symmetry group for meshes on
     orientation space

OUTPUT:

rmesh is a MeshStructure,
      it is derived from the input mesh by subdividing each simplicial element
      into a number of subelements based on the subdivision parameter n

NOTES:

*  This routine only subdivides the simplex.  If the
   mesh points are mapped, as in sphere meshes, then that mapping needs to be
   applied after this routine.
```

SliceMesh

```
SliceMesh - Planar slice through a 3D mesh

USAGE:

sm = SliceMesh(m, p, n)

INPUT:

m is a MeshStructure,
  for a general mesh
p is a 3-vector,
  a point on the slice plane
n is a 3-vector,
  the normal to the slice plane; it does not need to be unit length

OUTPUT:

sm is a MeshStructure,
   on the 2D planar section
smels is an integer array,
   the list of elements containing the nodes in ‘sm’, relative to the original
   mesh, ‘m’
smecrd is 4 x n,
   the barycentric (elemental) coordinates of the nodes in ‘sm’ relative to the
   original mesh, ‘m’
```

SmoothFunction

```
SmoothFunction - Smooth by convolution.

USAGE:

fsm = SmoothFunction(f, mesh, gqrule, @SmoothFun)
fsm = SmoothFunction(f, mesh, gqrule, @SmoothFun, arg1, ...)
```

```
INPUT:

f      is an n-vector,
       the values of a function on the mesh at the set of independent nodal
       points
mesh   is a MeshStructure,
       for any region
qpts   is d x n,
       the barycentric coordinates of the quadrature points
gqrule is a QRuleStructure,
       the global quadrature rule for the mesh

SmoothFun is a function handle,
          which defines the function to use for smoothing; it has the form:

          SmoothFun(center, points, ...)

          center is m x 1,
                 the center of the distribution
          points is m x n,
                 the list of points to evaluate

Other arguments are passed to ‘SmoothFun’

OUTPUT:

fsm is 1 x n,
   the nodal point values of the smoothed function at the set of independent
   nodal points

NOTES:

*  The function is smoothed by convolving a fixed distribution
   (e.g. Gaussian) with the given function. The result at each nodal point
   value is the integral of the input function times the smoothing function
   centered at that nodal point.

*  The result is not normalized.
```

SpreadRefPts

```
SpreadRefPts - Spread reference coordinates to all elements.

USAGE:

pts = SpreadRefPts(mesh, ref)

INPUT:

mesh is a MeshStructure
     with simplicial element type
ref is e x k,
    a list of barycentric coordinates for points in the reference element;

OUTPUT:
```

```
pts is d x k x m,
    the spatial coordinates of the reference points under the isoparametric
    mapping defined by the mesh; ‘d’ is the dimension of the mapped space, ’k’
    is the number of reference points, and ’m’ is the number of elements in
    ‘mesh’
```

NOTES:

* Typically this would be used for numerical quadrature. * e = d + 1 is not
  necessary; cross-dimension mappings are ok.

SubdivideSimplex

SubdivideSimplex - Regular subdivision of a simplex.

USAGE:

```
ref = SubdivideSimplex(dim, n)
[ref, con] = SubdivideSimplex(dim, n)
```

INPUT:

```
dim is a positive integer,
    the dimension of the simplex
n   is a positive integer,
    the number of subdivisions in each direction
```

OUTPUT:

```
ref is d x m,
    the list of barycentric coordinates of the points in the subdivision
con is d x e, (integer)
    the connectivity of the subdivision
```

ToAllNodes

ToAllNodes - Spread array at independent nodes to all nodes.

USAGE:

```
all = ToAllNodes(red, eqv)
```

INPUT:

```
red is m x n,
    a list of n m-vectors at the independent nodes of a mesh

eqv is 2 x l, (integer)
    the list of node equivalences (new #, old #)
```

OUTPUT:

```
all is m x k,
   the array at all nodes of the mesh; k = n + 1
```

3.3 Graphics

PlotFR

```
PlotFR - Plot a function on a fundamental region.

USAGE:

PlotFR(mesh, odf)
PlotFR(mesh, odf 'param', 'value', ...)

INPUT:

mesh is a MeshStructure:
        describing the fundamental region to be plotted
odf  is a vector of reals:,
        it contains nodal point values of the function to be plotted; the
        values are expected only for the independent nodes of the mesh

These arguments can be followed by a list of parameter/value pairs which
control certain plotting features. Options are:

'Symmetries'       string indicating symmetries of the FR
                   {'cubic'} | 'hexagonal'
'ShowMesh'         on | {off}
                   to show mesh lines on the surface plot
'Colormap'         ncolors x 3  (RGB)
                   to specify the figure's colormap; the default colormap is a
                   brightened version of `jet'
'BrightenColormap' scalar between 0 and 1 (default=0)
                   factor for brightening the colormap; a value of 0 does
                   nothing, a value of 1 makes everything white
'NumberOfColors'   positive integer (default = 64)
                   number of colors to use in default colormap

OUTPUT:  none

NOTES:

*  This function makes a figure containing side-by-side
   plots of the surface of the given fundamental region and of slices through
   it, with a common colorbar in the middle.
```

PlotFRPerimeter

```
PlotFRPerimeter - Plot perimeter of fundamental region.

USAGE:

perim =  PlotFRPerimeter(symtype)
```

```
INPUT:

symtype is a string,
        indicating the fundamental region type; possible values are:
        'cubic'|'hexagonal'; if the symmetry type is not recognized, a warning
        is issued

OUTPUT:

perim is a column vector of line handles:
         it contains handles to the lines outlining each face, as returned by
         the matlab 'plot3' function

NOTES:

* set 'hold' to on
```

PlotFRSlices

```
PlotFRSlices - Plot slices of the fundamental region.

USAGE:

PlotFRSlices(mesh, odf, type)
PlotFRSlices(mesh, odf, type, planes)

INPUT:

mesh is a MeshStructure,
     on the given fundamental region
odf  is a vector,
     the odf values at the independent nodal points of 'mesh'
type is a string, (optional, default: 'cubic')
     indicating the symmetry type; possible values are 'cubic' | 'hexagonal';
     the slices to display are based on the type
planes is an array of structures, (optional)
     each structure is expected to have components 'Point' and 'Normal'
     characterizing a plane; if specified, this overrides the 'type' input

OUTPUT:  none
```

PlotSphere

```
PlotSphere - Plot a function (pole figure) on the Sphere

USAGE:

PlotSphere(smesh, pf)
PlotSphere(smesh, pf, 'param', 'value', ...)

INPUT:

smesh is a MeshStructure,
      on the sphere (the usual one)
pf    is a vector,
      the nodal point values of thefunction (pole figure) to plot

These arguments can be followed by a list of parameter/value pairs which
control certain plotting features. Options are:

'ShowMesh'         on|{off}
                   to show mesh lines
'ShowQuadrants'    {on}|off
                   to show quadrant divisions on the sphere
'UpperHemisphere'  on|{off}
                   to limit quadrant divisions to only the upper hemisphere;
                   this only applies if 'ShowQuadrants' is in effect
'Colormap'         ncolors x 3  (RGB)
                   to specify the figure's colormap; the default colormap is a
                   brightened version of `jet'
'ShowColorBar'     on|{off}
                   to display a colorbar or not

OUTPUT:  none

NOTES:

*  `hold on' is in effect following this routine
```

PlotSphereQuadrants

```
PlotSphereQuadrants - Outline quadrants on unit sphere

USAGE:

PlotSphereQuadrants(ndiv)
PlotSphereQuadrants(ndiv, upper)

INPUT:

ndiv  is a positive integer,
      the number of points per circle
upper is a positive integer, (optional, default=0)
      if nonzero, only the upper half is drawn

OUTPUT:  none
```

PlotSurface

```
PlotSurface - Plot a function on surface in 3d.

USAGE:

PlotSurface(smesh, sfun)
PlotSurface(smesh, sfun, 'param', 'value', ...)

INPUT:
```

```
smesh is a MeshStructure,
      on a 2D surface in 3D
sfun  is a vector,
      the values on the nodes of the surface mesh

These arguments can be followed by a list of parameter/value pairs which
control certain plotting features. Options are:

'ShowMesh'    on|{off}
              to show the edges of the mesh

OUTPUT:  none

NOTES:

*  This routine calls `trimesh' with interpolated face color.
```

3.4 MatArray

DetMatArray

```
DetMatArray - Evaluate determinants for an array of matrices.

USAGE:

determ = DetMatArray(marray)

INPUT:

marray is n x n x l,
       an array of l n x n matrices

OUTPUT:

determ is 1 x l,
       the determinant of each matrix
```

InvMatArray

```
InvMatArray - Inverses for an array of matrices.

USAGE:

minv = InvMatArray(marray)

INPUT:

marray is n x n x l,
       an array of n x n matrices

OUTPUT:
```

```
minv is n x n x l,
     the inverse of each matrix
```

MultMatArray

```
MultMatArray - Multiply arrays of matrices.

USAGE:

prod = MultMatArray(ma1, ma2)

INPUT:

ma1 is m x n x l,
    an array of m x n matrices
ma2 is n x k x l,
    an array of n x k matrices

OUTPUT:

prod is m x k x l,
     the array of matrix products
```

SparseOfMatArray

```
SparseOfMatArray - Sparse matrix from array of matrices.

USAGE:

smat = SparseOfMatArray(marray)

INPUT:

marray is m x n x l,
       an array of m x n matrices

OUTPUT:

smat is l*m x l*n , (sparse)
     the sparse matrix form of the block matrix ‘marray’
```

3.5 Miscellaneous

Acknowledgments

```
Acknowledgments - Print acknowledgments.

USAGE:

Acknowledgments
```

```
a = Acknowledgments

INPUT:  none

OUTPUT:

a is a cell array of strings,
  containing the acknowledgments
```

OdfPfVersion

```
OdfPfVersion - Print version information.

USAGE:

OdfPfVersion;
info = OdfPfVersion;

INPUT:  none

OUTPUT:

info is a cell array of strings containing version information.
```

3.6 Misorientations

Misorientation

```
Misorientation - Return misorientation data for quaternions.

USAGE:

angle = Misorientation(q1, q2, sym)
[angle, mis] = Misorientation(q1, q2, sym)

INPUT:

q1 is 4 x n1,
   is either a single quaternion or a list of n quaternions
q2 is 4 x n,
   a list of quaternions

OUTPUT:

angle is 1 x n,
      the list of misorientation angles between q2 and q1
mis   is 4 x n, (optional)
      is a list of misorientations in the fundamental region (there are many
      equivalent choices)

NOTES:
```

```
*  The misorientation is the linear tranformation which
   takes the crystal basis given by q1 to that given by q2. The matrix of this
   transformation is the same in either crystal basis, and that is what is
   returned (as a quaternion). The result is inverse(q1) * q2. In the sample
   reference frame, the result would be q2 * inverse(q1). With symmetries, the
   result is put in the fundamental region, but not into the Mackenzie cell.
```

MisorientationStats

```
MisorientationStats - Misorientation correlation statistics.

USAGE:

stats = MisorientationStats(misorient, locations)

INPUT:

misorient is 4 x n,
          a list of misorientation quaternions, assumed to have been derived
          from properly clustered orientation data
locations is d x n, (d <= 3)
          a list of spatial locations corresponding to the misorientations

OUTPUT:

stats is a structure with five components:

      W is a 3 x 3 matrix (A in Barton paper) X is a d x d matrix (M in Barton
      paper) WX is a 3 x d matrix (cross-correlation
                 of normalized variables; X in Barton paper)
      wi is 3 x n, the unnormalized axial vectors xi is d x n, the unnormalized
      spatial directions
                      from the centroid

REFERENCE:

"A Methodology for Determining Average Lattice Orientation and Its Application
to the Characterization of Grain Substructure",

Nathan R. Barton and Paul R. Dawson,

Metallurgical and Materials Transactions A, Volume 32A, August 2001, pp.
1967--1975
```

3.7 PoleFigure

BuildOdfPfMatrix

```
BuildOdfPfMatrix - Build ODF/PF matrix in pieces.

USAGE:
```

```
opm = BuildOdfPfMatrix(hkl, mesh, sym, pts, div, invpf, block)

INPUT:

hkl,  mesh, sym, pts, div, invpf
      are the same as in ‘OdfPfMatrix’, except that ‘invpf’ is no longer
      optional
block is a postive integer,
      the number of pole figure points to be processed at a time; the pole
      figure points can be processed independently to control memory allocation

OUTPUT:

See ‘OdfPfMatrix’

NOTES:

*  See OdfPfMatrix for further documentation.

*  This is preferrd to OdfPfMatrix for problems with
   many divisions per fiber and many pole figure points.
```

FiberCoordinates

```
FiberCoordinates - Find mesh coordinates for a given fiber.

USAGE:

[fele, fcrd] = FiberCoordinates(fib, mesh)

INPUT:

fib is a 3 x ndiv x npole,
    it is an array with each page giving the fiber over the n’th pole point;
    usually the result of FiberOfPoint
mesh is a MeshStructure
    on the fundamental region

OUTPUT:

fele is ndiv x npole,
     the list of elements containing the fiber points.
fcrd is 4 x ndiv x npole,
     the barycentric coordinates of the fiber points

  Notes:

  * The call to the matlab builtin ‘tsearchn’ can fail, possibly due to the
     fact that the meshes are not necessarily Delaunay tesselations.
```

FiberOfPoint

```
FiberOfPoint - Find fiber above point for specified pole figure.
```

```
USAGE:

fib = FiberOfPoint(p, h, ndiv, qsym)
fib = FiberOfPoint(p, h, ndiv, qsym, invfib)

INPUT:

p    is 3 x n,
     an array of points on the sphere
h    is 3 x 1,
     a specified pole direction; if ‘invfib’ is nonzero, this is interpreted as
     a crystal direction; otherwise it is interpreted as a sample direction
ndiv is a positive integer,
     the number of equally spaced divisions along the fiber
qsym is 4 x m,
     the quaternion representation of the symmetry group
invfib is a scalar, (optional)
     a flag which, if nonzero, produces the inverse fiber

OUTPUT:

fib is 3 x ndiv x n,
    the column represents the Rodrigues vector of a point on the fiber, the row
    spans the points on the fiber, and the page spans the points on the sphere

NOTES:

* ‘h’ need not be a unit vector; it is normalized here
```

FindFiber

```
FindFiber - Find fiber for given crystal and sample directions.

USAGE:

fibs = FindFiber(c, s, ndiv)

INPUT:

c is 3 x m,
     an array of crystal directions
s is 3 x n,
     an array of sample directions; if n > 1 then m must be 1, and vice versa
ndiv is a postive integer,
     the number of points to return per fiber

OUTPUT:

fibs is 4 x ndiv x nfibs,
     an array of equally spaced points (quaternions) along each specified
     fiber; ‘nfibs’ is either ‘m’ or ‘n’ above

NOTES:

* The (c, s) fiber is the collection of rotations R
  such that Rc = s
```

```
* This routine can be called with many crystal directions
  and a single sample (as for inverse pole figures) or with many sample
  directions and a single crystal direction (as for pole figures)
```

MeshCoordinates

```
MeshCoordinates - find elements and barycentric coordinates of points

USAGE:

[els, crds] = MeshCoordinates(mesh, pts)

INPUT:

mesh is a MeshStructure,
     it should be Delaunay, but it may be okay anyway if it is not too irregular
pts  is m x n,
     a list of n m-dimensional points

OUTPUT:

els is an n-vector, (integers)
    the list of element numbers containing each point
crds is (m+1) x n,
    the barycentric coordinates of each point

NOTES:

*  The call to the matlab builtin ‘tsearchn’ can fail if
   the mesh is not Delaunay.
```

OdfPfMatrix

```
OdfPfMatrix - Find matrix relating ODF to pole figure.

USAGE:

opm = OdfPfMatrix(hkl, mesh, sym, pts, div)
opm = OdfPfMatrix(hkl, mesh, sym, pts, div, invpf)

INPUT:

hkl   is a 3-vector,
      the crystal direction specifying the pole figure
mesh  is a MeshStructure,
      on orientation space
sym   is 4 x s,
      the symmetry group in quaternions
pts   is 3 x p,
      a list of p points on the sphere (S^2)
div   is a positive integer,
      the number of divisions per fiber to use in calculating the fiber integral
invpf is a scalar, (optional, default = 0)
```

```
      a nonzero value flags causes computation of inverse pole figure matrix in
      which the ‘hkl’ is interpreted as a fixed sample direction and ‘pts’ is
      an array of crystal directions

OUTPUT:

opm is p x n, (sparse)
    the matrix which takes nodal point values on the fundamental region to pole
    figure or inverse values at the specified points

NOTES:

*  opm acts on the "reduced" set of nodes, the set of nodes
   in which equivalent nodes are combined into a single degree of freedom
*  this routine can be very memory intensive; you may need to
   build the matrix in pieces; if you have a lot of points and a large value of
   ‘div’, then you should break the points into smaller groups
*  the ODF-PF matrix preserves mean value, so that
   an ODF in MUD (multiples of uniform) maps to a pole figure also in MUD.
```

QFiberOfPoint

```
QFiberOfPoint - Find fiber above point in quaternions.

USAGE:

qfib = QFiberOfPoint(p, h, ndiv, qsym)
qfib = QFiberOfPoint(p, h, ndiv, qsym, invfib)

INPUT:

p    is 3 x n,
     an array of points on the sphere
h    is 3 x 1,
     a specified pole direction; if ‘invfib’ is nonzero, this is interpreted as
     a crystal direction; otherwise it is interpreted as a sample direction
ndiv is a positive integer,
     the number of equally spaced divisions along the fiber
qsym is 4 x m,
     the quaternion representation of the symmetry group
invfib is a scalar, (optional, default: 0)
     a flag which, if nonzero, produces the inverse fiber

OUTPUT:

qfib is 4 x ndiv x n,
     each column gives the quaternion representation of a point on the fiber;
     the row spans the points on the fiber; and the page (third index) spans
     the points on the sphere

NOTES:

* h need not be a unit vector; it is normalized here
```

3.8 Polytope

PolytopeStructure

```
PolytopeStructure - Structure for polytope.

USAGE:

pstruct = PolytopeStructure(matrix, rhs)
pstruct = PolytopeStructure(matrix, rhs, vertices)
pstruct = PolytopeStructure(matrix, rhs, vertices, faces)

INPUT:

matrix   is a m x d real array:
         the matrix of the linear inequalities defining the polytope (A, where
         Ax <= b)
rhs      is a m x 1 real array:
         the right hand side of the linear inequalities defining the polytope
         (b, where Ax <= b)
vertices is a d x n real array:  (optional)
         the list of n vertices in R^d
faces    is a 1 x f cell array:  (optional)
         each cell contains a list of vertex numbers for a given face (3D
         polytopes only)

OUTPUT:

pstruct is a Polytopestructure,
        it consists of two primary fields, .matrix and .rhs; optionally, it can
        have two more fields, .vertices and .faces; the optional fields are not
        computed here, and they are primarily used by the graphics functions
```

3.9 Rotations

BungeOfKocks

```
BungeOfKocks - Bunge angles from Kocks angles.

USAGE:

bunge = BungeOfKocks(kocks, units)

INPUT:

kocks is 3 x n,
      the Kocks angles for the same orientations
units is a string,
      either 'degrees' or 'radians'

OUTPUT:

bunge is 3 x n,
      the Bunge angles for n orientations

NOTES:

*  The angle units apply to both input and output.
```

BungeOfRMat

```
BungeOfRMat - Bunge angles from rotation matrices.

USAGE:

bunge = BungeOfRMat(rmat, units)

INPUT:

rmat  is 3 x 3 x n,
      an array of rotation matrices
units is a string,
      either 'degrees' or 'radians' specifying the output angle units

OUTPUT:

bunge is 3 x n,
      the array of Euler angles using Bunge convention
```

CubBaseMesh

```
  CubBaseMesh - Return base mesh for cubic symmetries

USAGE:

m = CubBaseMesh

INPUT:  no inputs

OUTPUT:

m is a MeshStructure,
     on the cubic fundamental region
```

CubPolytope

```
CubPolytope - Polytope for cubic fundamental region.

USAGE:

cubp = CubPolytope

INPUT:  none

OUTPUT:
```

```
cubp is a PolytopeStructure:
     it gives the polytope for the cubic fundamental region including the
     vertex list and the faces component (for plotting)
```

CubSymmetries

```
CubSymmetries - Return quaternions for cubic symmetry group.

USAGE:

csym = CubSymmetries

INPUT:  none

OUTPUT:

csym is 4 x 24,
     quaternions for the cubic symmetry group
```

HexBaseMesh

```
  HexBaseMesh - Return base mesh for hexagonal symmetries

USAGE:

m = HexBaseMesh

INPUT:  none

OUTPUT:

m is a MeshStructure,
  on the hexagonal fundamental region
```

HexPolytope

```
HexPolytope - Polytope for hexagonal fundamental region.

USAGE:

hexp = HexPolytope

INPUT:  none

OUTPUT:

hexp is a PolytopeStructure:
     the polytope of the hexgonal fundamental region; it includes the vertex
     list and the list of polygonal faces
```

HexSymmetries

```
HexSymmetries - Quaternions for hexagonal symmetry group.

USAGE:

hsym = HexSymmetries

INPUT:  none

OUTPUT:

hsym is 4 x 12,
     it is the hexagonal symmetry group represented as quaternions
```

KocksOfBunge

```
KocksOfBunge - Kocks angles from Bunge angles.

USAGE:

kocks = KocksOfBunge(bunge, units)

INPUT:

bunge is 3 x n,
      the Bunge angles for n orientations
units is a string,
    either 'degrees' or 'radians'

OUTPUT:

kocks is 3 x n,
      the Kocks angles for the same orientations

NOTES:

*  The angle units apply to both input and output.
```

QuatOfAngleAxis

```
QuatOfAngleAxis - Quaternion of angle/axis pair.

USAGE:

quat = QuatOfAngleAxis(angle, rotaxis)

INPUT:

angle is an n-vector,
      the list of rotation angles
raxis is 3 x n,
      the list of rotation axes, which need not be normalized (e.g. [1 1 1]'),
      but must be nonzero
```

```
OUTPUT:

quat is 4 x n,
     the quaternion representations of the given rotations. The first component
     of quat is nonnegative.
```

QuatOfRod

```
QuatOfRod - Quaternion from Rodrigues vectors.

USAGE:

quat = QuatOfRod(rod)

INPUT:

rod  is 3 x n,
     an array whose columns are Rodrigues parameters

OUTPUT:

quat is 4 x n,
     an array whose columns are the corresponding unit quaternion parameters;
     the first component of 'quat' is nonnegative
```

QuatProd

```
QuatProd - Product of two unit quaternions.

USAGE:

 qp = QuatProd(q2, q1)

INPUT:

 q2, q1 are 4 x n,
        arrays whose columns are quaternion parameters

OUTPUT:

 qp is 4 x n,
    the array whose columns are the quaternion parameters of the product; the
    first component of qp is nonnegative

 NOTES:

 *  If R(q) is the rotation corresponding to the
    quaternion parameters q, then

    R(qp) = R(q2) R(q1)
```

RMatOfBunge

```
RMatOfBunge - Rotation matrix from Bunge angles.

USAGE:

rmat = RMatOfBunge(bunge, units)

INPUT:

bunge is 3 x n,
      the array of Bunge parameters
units is a string,
      either 'degrees' or 'radians'

OUTPUT:

rmat is 3 x 3 x n,
     the corresponding rotation matrices
```

RMatOfQuat

```
RMatOfQuat - Convert quaternions to rotation matrices.

USAGE:

rmat = RMatOfQuat(quat)

INPUT:

quat is 4 x n,
     an array of quaternion parameters

OUTPUT:

rmat is 3 x 3 x n,
     the corresponding array of rotation matrices

NOTES:

*  This is not optimized, but still does okay
   (about 6,700/sec on intel-linux ~2GHz)
```

RodDifferential

```
RodDifferential - Differential map for Rodrigues

USAGE:

diff = RodDifferential(rmesh, refpts)

INPUT:

mesh   is a mesh,
       on a Rodrigues mesh
refpts is 4 x n,
       a list of points in the reference element, usually the quadrature
       points, given in barycentric coordinates

OUTPUT:

diff is 4 x 3 x nq,
     a list of tangent vectors at each reference point for each element; nq is
     the number of global quadrature points, that is n x ne, where ne is the
     number of elements
```

RodDistance

```
RodDistance - Find angular distance between rotations.

USAGE:

dist = RodDistance(pt, ptlist, sym)

INPUT:

pt     is 3 x 1,
       a point given in Rodrigues parameters
ptlist is 3 x n,
       a list of points, also Rodrigues
sym    is 4 x m,
       the symmetry group in quaternions

OUTPUT:

dist   is 1 x n,
       the distance between ‘pt’ and each point in ‘ptlist’
```

RodGaussian

```
RODGAUSSIAN - Gaussian distribution on angular distance.

USAGE:

gauss = RodGaussian(cen, pts, stdev, sym)

INPUT:

cen   is 3 x 1,
      the center of the distribution (in Rodrigues parameters)
pts   is 3 x n,
      a list of points (Rodrigues parameters)
stdev is 1 x 1,
      the standard deviation of the distribution
sym   is 4 x k,
      the symmetry group (quaternions)

OUTPUT:
```

```
gauss is 1 x n,
      the list of values at each input point

NOTES:

*  This returns the values of a (not normalized) 1D Gaussian
   applied to angular distance from the center point
*  The result is not normalized to have unit integral.
```

RodMetric

```
RodMetric - find volume integration factor due to metric

USAGE:

metric = RodMetric(rod)

INPUT:

rod is 3 x n,
    an array of 3-vectors (Rodrigues parameters)

OUTPUT:

metric is 1 x n,
       the metric at each point of ‘rod’
```

RodOfQuat

```
RodOfQuat - Rodrigues parameterization from quaternion.

USAGE:

rod = RodOfQuat(quat)

INPUT:

quat is 4 x n,
     an array whose columns are quaternion paramters; it is assumed that there
     are no binary rotations (through 180 degrees) represented in the input
     list

OUTPUT:

  rod is 3 x n,
   an array whose columns form the Rodrigues parameterization of the same
   rotations as quat
```

ToFundamentalRegion

```
ToFundamentalRegion - Put rotation in fundamental region.
```

```
USAGE:

rod = ToFundamentalRegion(quat, qsym)

INPUT:

quat is 4 x n,
     an array of n quaternions
qsym is 4 x m,
     an array of m quaternions representing the symmetry group

OUTPUT:

rod is 3 x n,
    the array of Rodrigues vectors lying in the fundamental region for the
    symmetry group in question

NOTES:

*  This routine is very memory intensive since it
   applies all symmetries to each input quaternion.
```

ToFundamentalRegionQ

```
ToFundamentalRegionQ - To quaternion fundamental region.

USAGE:

q = ToFundamentalRegionQ(quat, qsym)

INPUT:

quat is 4 x n,
     an array of n quaternions
qsym is 4 x m,
     an array of m quaternions representing the symmetry group

OUTPUT:

q is 4 x n, the array of quaternions lying in the
            fundamental region for the symmetry group in question

NOTES:

*  This routine is very memory intensive since it
   applies all symmetries to each input quaternion.
```

3.10 Sphere

PSphDistance

```
PSphDistance - Distance on projective sphere.
```

```
USAGE:

dist = PSphDistance(pt, ptlist)

INPUT:

pt     is 3 x 1,
       a point on the unit sphere (S^2)
ptlist is 3 x n,
       a list of points on the unit sphere

OUTPUT:

dist is 1 x n,
     the distance from ‘pt’ to each point in the list

NOTES:

*  The distance between two points on the sphere is the angle
   in radians between the two vectors. On the projective sphere, antipodal
   points are considered equal, so the distance is the minimum of the distances
   obtained by including the negative of pt as well.
```

PSphGaussian

```
PSphGaussian - Gaussian distribution for smoothing on projective sphere.

USAGE:

fsm = PSphGaussian(center, pts, stdev)

INPUT:

center is 3 x 1,
       the center of the distribution
pts    is 3 x n,
       a list of points on the sphere; antipodal points are considered equal
stdev  is 1 x 1,
       the (1D) standard deviation

OUTPUT:

fsm is 1 x n,
    the list of values at each point of pts

Notes:

*  The result is not normalized, so this may have to be
   done after the fact.
*  The distribution is a 1-D normal distribution applied
   to the distance function on the projective sphere.
*  The actual scaling factor to give unit integral over
   the projective sphere is not computed; the result is not normalized.
```

SphBaseMesh

```
SphBaseMesh - Generate base mesh for spheres.

USAGE:

mesh = SphBaseMesh(dim)
mesh = SphBaseMesh(dim, 'param', 'value')

INPUT:

dim is a positive integer,
    the dimension of the sphere (2 for the usual sphere S^2)

These arguments can be followed by a list of parameter/value pairs which
specify keyword options. Available options include:

'Hemisphere'  'on'|'off'
              to mesh only the upper hemisphere


OUTPUT:

mesh is a MeshStructure,
     on the sphere of the specified dimension

NOTES:
```

SphCrdMesh

```
SphCrdMesh - Generate a hemisphere mesh based on spherical coordinates.

USAGE:

smesh = SphCrdMesh(ntheta, nphi)

INPUT:

ntheta is a positive integer,
       the number of subdivisions in theta
nphi   is a positive integer,
       the number of subdivisions in phi

OUTPUT:

smesh is a MeshStructure,
      on the hemisphere (H^2)
```

SphDifferential

```
SphDifferential - Compute differential of mapping to sphere.

USAGE:
```

```
diff = SphDifferential(mesh, refpts)

INPUT:

mesh   is a mesh,
       on a sphere of any dimension
refpts is d x n,
       a list of points in the reference element, usually the quadrature
       points, given in barycentric coordinates

OUTPUT:

diff is d x (d-1) x nq,
     a list of tangent vectors at each reference point for each element; nq is
     the number of global quadrature points, that is n x ne, where ne is the
     number of elements
```

SphDistance

```
SphDistance - Find distance on sphere between a fixed point and others.

USAGE:

dist = SphDistance(pt, ptlist)

INPUT:

pt     is 3 x 1,
       a point on the unit sphere (S^2)
ptlist is 3 x n,
       a list of points on the unit sphere (S^2)

OUTPUT:

dist is 1 x n,
     the distance from ‘pt’ to each point on the list

NOTES:

*  The distance is just the angle between the two vectors.
```

SphDistanceFunc

```
SphDistanceFunc - Return half sum of squared distances on sphere.

USAGE:

f           = SphDistanceFunc(x, pts, @Sofx)
[f, gf]     = SphDistanceFunc(x, pts, @Sofx)
[f, gf, Hf] = SphDistanceFunc(x, pts, @Sofx)

INPUT:

x    is d x 1,
     a point in parameter space
pts  is (d+1) x n,
     a list of n points on the sphere
Sofx is a function handle,
     returning parameterization component quantities (function, gradient and
     Hessian)

OUTPUT:

f  is a scalar,
   the objective function at x
gf is a vector,
   the gradient of f at x
Hf is a matrix,
   the Hessian of f at x

NOTES:

*  See MisorientationStats
```

SphGQRule

```
SphGQRule - Global quadrature rule for sphere.

USAGE:

[gqrule, l2ip] = SphGQRule(mesh, qrule)

INPUT:

mesh  is a MeshStructure,
      on a sphere of any dimension
qrule is a QRuleStructure,
      on the reference element of the mesh

OUTPUT:

gqrule is a QRuleStructure,
       for the entire mesh
l2ip   is n x n, (sparse)
       it gives the l2 inner product in terms of function values at the nodal
       points
```

SphGaussian

```
SphGaussian - Gaussian distribution on angular distance.

USAGE:

fsm = SphGaussian(cen, pts, stdev)

INPUT:

cen   is 3 x 1,
      the center of the distribution
pts   is 3 x n,
      a list of points on the sphere at which to evaluate the result
stdev is a scalar,
      the standard deviation

OUTPUT:

fsm is 1 x n,
    the list of values at each point of ‘pts’

NOTES:

*  This returns the values of a 1D Gaussian applied
   to spherical distance from the center point
*  The result is not normalized to have unit integral
   over the sphere.
```

SphH1SIP

```
SphH1SIP -- H^1 semi-inner product on sphere

USAGE:

h1sip = SphH1SIP(mesh, qrule)

INPUT:

mesh  is a MeshStructure,
      a mesh on a sphere of any dimension
qrule is a QRuleStructure,
      a quadrature rule for the reference simplex

OUTPUT:

h1sip is n x n, (sparse)
      it is the matrix of the H^1 semi-inner product, where n is the number of
      independent degrees of freedom associated with the mesh
```

SphereAverage

```
SphereAverage - Find ‘average’ of list of points on sphere.

USAGE:

avg = SphereAverage(pts)
[avg, optdat] = SphereAverage(pts, Pzation, nlopts)

INPUT:

pts     is m x n,
        a list of n points in R^m of unit length
Pzation is a function handle,
        (see note below)
```

```
x0      is the initial guess,
        in given parameterization
nlopts  are options to be passed to the nonlinear minimizer.

OUTPUT:

avg is m x 1,
       is a unit vector representing the "average" of ‘pts’
optdat is a cell array,
       with three members, {fval, exitflag, output} (see documentation for
       ‘fminunc’)

NOTES:

*  If only one argument is given, the average returned is
   the arithmetic average of the points. If all three arguments are given, then
   the average is computed using unconstrained minimization of the sum of
   squared angles from the data points, using the parameterization specified
   and the options given.

*  See the matlab builtin ‘fminunc’ for details.

*  This routine needs to be fixed.  Currently it uses the
   parameterization given by ‘SpherePZ’ instead of the function handle
   ‘PZation’.
```

SpherePZ

```
SpherePZ - Generate point, gradients and Hessians of map to sphere.

USAGE:

sk           = SpherePZ(x)
[sk, gk]     = SpherePZ(x)
[sk, gk, Hk] = SpherePZ(x)

INPUT:

x is d x 1,
  a vector with norm <= 1

OUTPUT:

sk is e x 1,
   a point on the sphere (sqrt(1-x^2), x)
gk is d x e,
   the gradients of each component of sk
Hk is d x d x e,
   the Hessians of each component of sk
```

XYZOfThetaPhi

```
XYZOfThetaPhi - Map spherical coordinates to sphere.
```

```
USAGE:

xyz = XYZOfThetaPhi(thetaphi)

INPUT:

thetaphi is 2 x n,
         the spherical coordinates for a list of n points; theta is the angle
         that the projection onto x-y plane makes with the x-axis, and phi is
         the angle with z-axis

OUTPUT:

xyz is 3 x n,
    the Cartesian coordinates of the points described by (theta, phi)

NOTES:

*  The matlab builtin ‘sph2cart’ could also be used, but
   the convention for phi is different (90 degrees minus thisone).
```

3.11 Utility

AggregateFunction

```
AggregateFunction - Create a function from an aggregate of points.

USAGE:

aggf = AggregateFunction(pts, agg, wts, @PointFun)
aggf = AggregateFunction(pts, agg, wts, @PointFun, pfarg1, ...)

INPUT:

pts is d x n,
    the set of points on which to evaluate the aggregate function
agg is d x m,
    a collection of points (the aggregate)
wts is 1 x m,
    the weighting associated with points in ‘agg’
PointFun is a function handle,
    the function which evaluates the distribution associated with each point of
    the aggregate; the required interface to PointFun is:

    PointFun(center, points [, args])
             center is d x 1, the center of the distribution points is d x n, a
             list of points to evaluate args are function-specific arguments

Remaining arguments are passed to PointFun.

OUTPUT:

aggf is 1 x n,
     the values of the aggregate function at each point in ‘pts’;
```

```
NOTES:

* Each point in the aggregate is the center of a distribution
  over the whole space, given by PointFun; all of these distributions are
  superposed to give the resulting aggregate function, which is then evaluated
  at the specified point.
```

CycleIndices

```
CycleIndices - Cycle the indices 1:n.

USAGE:

cycle = CycleIndices(n)

INPUT:

n is a postive integer

OUTPUT:

cycle is an n x n array of integers ,
      the columns are the indices 1:n cyclically permuted
```

MetricGij

```
MetricGij - Compute components of metric from differential.

USAGE:

gij = MetricGij(diff)

INPUT:

diff is m x n x l,
     the array of n tangent vectors of dimension m at each of l points

OUTPUT:

gij is n x n x l,
    the metric components (dot(ti, tj)) at each of the l points
```

OnOrOff

```
OnOrOff - Convert on|off string to 1|0 integer.

USAGE:

toggle = OnOrOff(string)

INPUT:
```

```
string is either 'on' or 'off' (case is ignored)

OUTPUT:

toggle is a scalar,
       0   if string matches 'off' 1   if string matches 'on'
```

OptArgs

```
OptArgs - Set options from cell array.

USAGE:

opts = OptArgs(optkeys, optargs)

INPUT:

optkeys is a cell array, (of length 2*n)
        consisting of data in the form {string1, object1, string2, object2,
        ...}; the strings are the parameter names and objects are the default
        values associated with each paramter
optargs is a cell array, (of length 2*m)
        it is of the same form as `optkeys'; its values override the defaults

OUTPUT:

opts is a structure,
        the fields are parameter names from `optkeys', and the values are
        either the default value or the overriding value from `optargs'

NOTES:

*  Values in `optkeys' which are cell arrays need to be
   contained inside another cell array; see `struct' for details, whereas
   values in `optargs' which are cell arrays should not be placed in another
   cell array. The difference arises because the resulting structure is
   generated by a call to the matlab `struct' command with argument `optkeys',
   but the structure fields are updated one by one using the values in
   `optargs'.

*  The general usage of this function would be to set optkeys
   in the user function and pass varargin as optargs to update the default
   values.
```

RankOneMatrix

```
RankOneMatrix - Create rank one matrices (dyadics) from vectors.

USAGE:

r1mat = RankOneMatrix(vec1)
r1mat = RankOneMatrix(vec1, vec2)

INPUT:
```

```
vec1 is m1 x n,
     an array of n m1-vectors
vec2 is m2 x n, (optional)
     an array of n m2-vectors

OUTPUT:

r1mat is m1 x m2 x n,
      an array of rank one matrices formed as c1*c2' from columns c1 and c2

With one argument, the second vector is taken to the same as the first.

NOTES:

*  This routine can be replaced by MultMatArray.
```

UniqueVectors

```
UniqueVectors - Remove near duplicates from a list of vectors.

USAGE:

[uvec, ord, iord] = UniqueVectors(vec)
[uvec, ord, iord] = UniqueVectors(vec, tol)

INPUT:

vec is d x n,
    an array of n d-vectors
tol is a scalar, (optional)
    the tolerance for comparison; it defaults to 1.0e-14

OUTPUT:

uvec is d x m,
     the set of unique vectors; two adjacent vectors are considered equal if
     each component is within the given tolerance
ord  is an m-vector, (integer)
     which relates the input vector to the output vector, i.e. uvec = vec(:,
     ord)
iord is an n-vector, (integer)
     which relates the reduced vector to the original vector, i.e. vec =
     uvec(:, iord)

NOTES:

*  After sorting, only adjacent entries are tested for equality
   within the tolerance. For example, if x1 and x2 are within the tolerance,
   and x2 and x3 are within the tolerance, then all 3 will be considered the
   same point, even though x1 and x3 may not be within the tolerance.
   Consequently, if you make the tolerance too large, all the points will be
   considered the same. Nevertheless, this routine should be adequate for the
   its intended application (meshing), where the points fall into
   well-separated clusters.
```

UnitVector

```
UnitVector - Normalize an array of vectors.

USAGE:

uvec = UnitVector(vec)
uvec = UnitVector(vec, ipmat)

INPUT:

vec   is m x n,
      an array of n nonzero vectors of dimension m
ipmat is m x m, (optional)
      this is a (SPD) matrix which defines the inner product on the vectors by
      the rule:
         norm(v)^2 = v' * ipmat * v

      If `ipmat' is not specified, the usual Euclidean inner product is used.

OUTPUT:

uvec is m x n,
     the array of unit vectors derived from `vec'
```

Acknowledgments

The author wishes to acknowledge the contributions of several individuals. In particular, Ashish Kumar deserves special mention since his doctoral work provided the basis for our group's continuing use of finite elements on Rodrigues parameters. All members of the Cornell DPLab group proved helpful including P. Dawson, N. Barton and J. Bernier, in particular. Chris Pelkie from the Cornell Theory Center is of course an indispensible resource for visualization.

Principal funding has been provided by the Office of Naval Research, Structural Materials Program. Supplemental funding was provided under the DoD PET program.

Matlab® is a trademark of The MathWorks, Inc.

Thanks also to R. Cools for use of his nice website for numerical cubature(*www.cs.kuleuven.ac.be/ nines/ecf/*). Refer to "An encyclopaedia of cubature formulas", presented at the Oberwolfach meeting on "Numerical Integration and its Complexity", November 18-24, 2001.