Simple 1D FEM in Julia

David Bindel

Table of contents

Introduction

This document is automatically extracted from the sources found at https://github.com/bindel-group/simple_jfem. The code is written as a pedagogical example. It largely mirrors the structure of an analogous C code (https://github.com/bindel-group/simple_cfem).

Shape functions

A *shape function* on a reference domain is a basis function used for interpolation on that domain. We will generally use Lagrange shape functions (also called nodal shape functions), which are one at one nodal point in a reference domain and zero at the others. We want to be able to compute both the values of all shape functions at a point in the domain and also their derivatives (stored as a matrix with d rows for a d-dimensional reference domain).

1D building blocks

A useful building block is the Lagrange polynomials through 2, 3, or 4 equispaced points, which span the linear (P1), quadratic (P2), and cubic (P3) polynomial spaces on $[-1, 1]$, respectively.

```
shapes1dP1(x) = (0.5*(1-x), 0.5*(1+x))
dshapes1dP1(x) =(-0.5,      0.5)

shapes1dP2(x) = (-0.5*(1-x)*x, (1-x)*(1+x), 0.5*x*(1+x))
dshapes1dP2(x) =(-0.5*(1-2*x), -2*x,        0.5*(1+2*x))

function shapes1dP3(x)
```

```
    (-1.0/16) * (1-x)*(1-3*x)*(1+3*x),
    ( 9.0/16) * (1-x)*(1-3*x)*(1+x),
    ( 9.0/16) * (1-x)*(1+3*x)*(1+x),
    (-1.0/16) * (1-3*x)*(1+3*x)*(1+x)
end

function dshapes1dP3(x)
    1.0/16 * ( 1+x*( 18+x*-27)),
    9.0/16 * (-3+x*(-2+x* 9)),
    9.0/16 * ( 3+x*(-2+x*-9)),
    1.0/16 * (-1+x*( 18+x* 27))
end
```

Shape class macro

We make a general structure type for accessing shape functions. For a given set of shapes, we keep some storage for the shape functions (`N`) and their derivatives (`dN`). Because we are defining the same types of methods for each instance, we use Julia's macro facility to help write the routines for us.

```
abstract type ShapeFuns end

macro make_shape(classname, nfun, dim, body)
    quote
        struct $(esc(classname)) <: ShapeFuns
            N  :: Vector{Float64}  # Shape functions (nshapes)
            dN :: Matrix{Float64}  # Derivatives (nshapes-by-dshapes)
        end

        # Default constructor
        $(esc(classname))() =
            $(esc(classname))(zeros($nfun), zeros($dim, $nfun))

        # Get the number of shapes and dimension of the space
        $(esc(:nshapes))( :: $(esc(classname))) = $nfun
        $(esc(:dshapes))( :: $(esc(classname))) = $dim

        # Routine for actual computation (looks like a call to the object)
        function (s :: $(esc(classname)))(xx)
            $body
            s
```

```
        end
    end
end
```

1D shape functions

The 1D shape function classes just call the Lagrange polynomial support routines defined earlier. For each, shape, we also define a `refnodes` array with the list of reference-domain locations for the nodes.

```
@make_shape Shapes1dP1 2 1 begin
    x = xx[1]
    s.N[:]  .= shapes1dP1(x)
    s.dN[:] .= dshapes1dP1(x)
end
refnodes(::Shapes1dP1) = [-1.0 1.0]

@make_shape Shapes1dP2 3 1 begin
    x = xx[1]
    s.N[:]  .= shapes1dP2(x)
    s.dN[:] .= dshapes1dP2(x)
end
refnodes(::Shapes1dP2) = [-1.0 0.0 1.0]

@make_shape Shapes1dP3 4 1 begin
    x = xx[1]
    s.N[:]  .= shapes1dP3(x)
    s.dN[:] .= dshapes1dP3(x)
end
refnodes(::Shapes1dP3) = [-1.0 -1.0/3 1.0/3 1.0]
```

2D shape functions

The P1 and P2 shapes are just tensor products of the 1D P1 and P2 shapes. The serendipity element S2 is like the P2, but doesn't keep the "bubble mode" shape function associated with the center node.

```
@make_shape Shapes2dP1 4 2 begin
    Nx1,  Nx2  = shapes1dP1(xx[1])
    Ny1,  Ny2  = shapes1dP1(xx[2])
```

```
    dNx1, dNx2 = dshapes1dP1(xx[1])
    dNy1, dNy2 = dshapes1dP1(xx[2])
    s.N[:]  .= ( Nx1* Ny1,  Nx2* Ny1,  Nx2* Ny2,  Nx1* Ny2)
    s.dN[:] .= (dNx1* Ny1, Nx1*dNy1,
                dNx2* Ny1, Nx2*dNy1,
                dNx2* Ny2, Nx2*dNy2,
                dNx1* Ny2, Nx1*dNy2)
end
refnodes(::Shapes2dP1) =
    [-1.0   1.0   1.0  -1.0 ;
     -1.0  -1.0   1.0   1.0 ]

@make_shape Shapes2dP2 9 2 begin
    Nx1,  Nx2,  Nx3  = shapes1dP2(xx[1])
    Ny1,  Ny2,  Ny3  = shapes1dP2(xx[2])
    dNx1, dNx2, dNx3 = dshapes1dP2(xx[1])
    dNy1, dNy2, dNy3 = dshapes1dP2(xx[2])
    s.N[:] .=  ( Nx1* Ny1,  Nx2* Ny1,  Nx3* Ny1,
                 Nx3* Ny2,  Nx3* Ny3,  Nx2* Ny3,
                 Nx1* Ny3,  Nx1* Ny2,  Nx2* Ny2)
    s.dN[:] .= (dNx1* Ny1, Nx1*dNy1,
                dNx2* Ny1, Nx2*dNy1,
                dNx3* Ny1, Nx3*dNy1,
                dNx3* Ny2, Nx3*dNy2,
                dNx3* Ny3, Nx3*dNy3,
                dNx2* Ny3, Nx2*dNy3,
                dNx1* Ny3, Nx1*dNy3,
                dNx1* Ny2, Nx1*dNy2,
                dNx2* Ny2, Nx2*dNy2)
end
refnodes(::Shapes2dP2) =
    [-1.0   0.0   1.0   1.0   1.0   0.0  -1.0  -1.0   0.0 ;
     -1.0  -1.0  -1.0   0.0   1.0   1.0   1.0   0.0   0.0 ]

@make_shape Shapes2dS2 8 2 begin
    x, y = xx
    Nx1,  Nx2,  Nx3  = shapes1dP2(x)
    Ny1,  Ny2,  Ny3  = shapes1dP2(y)
    dNx1, dNx2, dNx3 = dshapes1dP2(x)
    dNy1, dNy2, dNy3 = dshapes1dP2(y)
    s.N[:] .=  ( Nx1* Ny1,  Nx2* Ny1,  Nx3* Ny1,
                 Nx3* Ny2,  Nx3* Ny3,  Nx2* Ny3,
```

```
                 Nx1* Ny3,  Nx1* Ny2)
    s.dN[:] .= (dNx1* Ny1, Nx1*dNy1,
                dNx2* Ny1, Nx2*dNy1,
                dNx3* Ny1, Nx3*dNy1,
                dNx3* Ny2, Nx3*dNy2,
                dNx3* Ny3, Nx3*dNy3,
                dNx2* Ny3, Nx2*dNy3,
                dNx1* Ny3, Nx1*dNy3,
                dNx1* Ny2, Nx1*dNy2)
end
refnodes(::Shapes2dS2) =
    [-1.0   0.0   1.0   1.0   1.0   0.0  -1.0  -1.0   0.0 ;
     -1.0  -1.0  -1.0   0.0   1.0   1.0   1.0   0.0   0.0 ]
```

Triangle shapes

For now, we only have linear shape functions on a triangle.

```
@make_shape Shapes2dT1 3 2 begin
    x, y = xx
    s.N[:]  .= (1.0-x-y, x, y)
    s.dN[:] .= (-1.0, -1.0,
                 1.0,  0.0,
                 0.0,  1.0)
end
refnodes(::Shapes2dT1) =
    [ 0.0 1.0 0.0 ;
      0.0 0.0 1.0 ]
```

Quadrature rules

Quadrature rules approximate integrals with formulas of the form

$$\int_\Omega f(x)\, d\Omega(x) \approx \sum_{j=1}^{p} f(\xi_j) w_j$$

where $\xi_j \in \Omega$ and $w_j \in \mathbb{R}$ are known as the quadrature nodes (or points) and weights, respectively.

A good source of quadrature rules for various domains can be found in Stroud's book on *Approximate calculation of multiple integrals* (Prentice Hall, 1971).

Gaussian-Legendre quadrature rules

Gauss-Legendre quadrature rules (sometimes just called Gauss quadrature rules when the context is clear) are p-point rules on $[-1, 1]$ that are characterized by the fact that they are exact when f is a polynomial of degree at most $2p - 1$.

Gauss-Legendre nodes are zeros of Legendre polynomials, while the weights can be computed via an eigenvalue decomposition (using the Golub-Welsch algorithm). However, we do not need very high-order quadrature rules, and so only provide nodes and weights for rules up to $p = 10$ (probably more than we need), which are tabulated in many places. Because this is just a table lookup, we don't bother to include the code in the automated documentation.

```
function gauss_point(i, npts)
    gauss_pts = (
        # ... the points
      )

    gauss_pts[( npts*(npts-1) )/2 + i]
end

function gauss_weight(i, npts)
    gauss_wts = (
        # ... and the weights
    )

    gauss_wts[( npts*(npts-1) )/2 + i]
end
```

Quadrature iterator interface

The `QuadratureRule` abstract type provides a base type for all quadrature rules. We assume that it provides methods

- `quad_npoints(rule)`: Returns the number of points
- `quad_dim(rule)`: Returns the dimension of the reference domain
- `quad_point(rule, i)`: Returns the quadrature point ξ_i
- `quad_weight(rule, i)`: Returns the quadrature weight w_i

One can also provide a `quad_pointwt` that returns the point and weight as a pair (by default, this just calls the individual `quad_point` and `quad_weight` methods.

For any quadrature rule, we overload the `Base.iterate` interface so that we can use it in `for` loops, writing expressions like

```
I = sum( f(xi) * wt for (xi, wt) in rule )
```

to approximate the integral via the rule.

```
abstract type QuadratureRule end

quad_pointwt(q :: QuadratureRule, i) =
    (quad_point(q, i), quad_weight(q, i))

Base.iterate(q :: QuadratureRule, state=1) =
    state > quad_npoints(q) ? nothing : (quad_pointwt(q, state), state+1)
```

1D Gauss quadrature interfaces

Our `GaussRule1d` rule just provides an alternate interface to the `gauss_point` and `gauss_weight` functions defined earlier

```
struct GaussRule1d <: QuadratureRule
    npts :: Int
end

quad_npoints(q :: GaussRule1d) = q.npts
quad_dim(q :: GaussRule1d) = 1
quad_point(q :: GaussRule1d, i) = gauss_point(i, q.npts)
quad_weight(q :: GaussRule1d, i) = gauss_weight(i, q.npts)
```

The `GaussRule1dv` rule returns the quadrature points in a vector of length 1 (rather than returning them as scalars)

```
struct GaussRule1dv <: QuadratureRule
    xi :: Vector{Float64}
    npts :: Int
end

GaussRule1dv(npts) = GaussRule1dv(zeros(1), npts)

quad_npoints(q :: GaussRule1dv) = q.npts
quad_dim(q :: GaussRule1dv) = 1

function quad_point(q :: GaussRule1dv, i)
```

```
    q.xi[1] = gauss_point(i, q.npts)
    q.xi
end

quad_weight(q :: GaussRule1dv, i) = gauss_weight(i, q.npts)
```

Product Gauss rules

A 2D tensor product Gauss rule for the domain $[-1,1]^2$ involves a grid of `npts1`-by-`npts1` quadrature points with coordinates given by 1D Gauss quadrature rules.

```
struct GaussRule2d <: QuadratureRule
    xi :: Vector{Float64}
    npts1 :: Int
 end

GaussRule2d(npts1) = GaussRule2d(zeros(2), npts1)

quad_npoints(q :: GaussRule2d) = q.npts1 * q.npts1
quad_dim(q :: GaussRule2d) = 2

function quad_point(q :: GaussRule2d, i)
    ix, iy = ((i-1)%q.npts1)+1, div(i-1,q.npts1)+1
    q.xi[:] .= (gauss_point(ix, q.npts1), gauss_point(iy, q.npts1))
    q.xi
end

function quad_weight(q :: GaussRule2d, i)
    ix, iy = ((i-1)%q.npts1)+1, div(i-1,q.npts1)+1
    gauss_weight(ix, q.npts1) * gauss_weight(iy, q.npts1)
end

function quad_pointwt(q :: GaussRule2d, i)
    ix, iy = ((i-1)%q.npts1)+1, div(i-1,q.npts1)+1
    q.xi[:] .= (gauss_point(ix, q.npts1), gauss_point(iy, q.npts1))
    wt = gauss_weight(ix, q.npts1) * gauss_weight(iy, q.npts1)
    (q.xi, wt)
end
```

Triangle mid-side rule

For a triangle, a rule based on the three mid-side values is exact for every polynomial with total degree less than or equal to 2 (which is enough for our purposes). This is sometimes called the Hughes formula.

```
struct HughesRule2d <: QuadratureRule
    xi :: Vector{Float64}
end

HughesRule2d() = HughesRule2d(zeros(2))

quad_npoints(q :: HughesRule2d) = 3
quad_dim(q :: HughesRule2d) = 2
quad_weight(q :: HughesRule2d, i) = 1.0/6

function quad_point(q :: HughesRule2d, i)
    pts = (0.5, 0.0,
           0.5, 0.5,
           0.0, 0.5)
    q.xi[:] .= pts[2*i-1:2*i]
end
```

Mapped subdomains

Our shape functions and quadrature rules are defined on simple reference domains. To accommodate more complicated domains, we consider mappings $\chi : \mathbb{R}^d \to \mathbb{R}^d$ from a reference domain Ω_0 into a spatial domain Ω_e (with $\chi(\Omega_0) = \Omega_e$). We will typically write $x = \chi(X)$ for generic points in the spatial and reference domains.

Mapping functions and derivatives

With the mapping between domains in hand, We can identify functions f^0 in the reference domain with functions f^e in the spatial domain:

$$f^e(x) = (f^0 \circ \chi^{-1})(x) = f^0(X)$$

and the derivative mapping is given by

$$f^{e\prime}(x) = f^{0\prime}(X) J(X)^{-1}$$

where $J(X) = \partial\chi/\partial X$ is the Jacobian of the reference-to-spatial mapping. In terms of gradients, we can write this as

$$\nabla_x f^e(x) = J(X)^{-T} \nabla_X f^0(X).$$

Because we expect to be computing these types of transformations a lot, we would rather not constantly allocate and deallocate space for LU factorization objects. Therefore, we define helper functions that directly call the LAPACK interfaces for LU factorization of J and subsequent solves with J^T (for a generic gradient g or for the shape function gradients stored in a `ShapeFuns` struct).

```
r2s_factor!(J, ipiv) = LAPACK.getrf!(J, ipiv)
r2s_gradients!(J, ipiv, g :: AbstractMatrix) = LAPACK.getrs!('T', J, ipiv, g)
r2s_gradients!(J, ipiv, s :: ShapeFuns) = r2s_gradients!(J, ipiv, s.dN)
```

We also define a function to get the determinant out of the LU factorization of J.

```
function r2s_det(J, ipiv)
    detJ = 1.0
    for k = 1:size(J,2)
        detJ *= J[k,k]
        if ipiv[k] ≠ k
            detJ = -detJ
        end
    end
    detJ
end
```

Mapped quadrature

We can also express integrals over Ω_e in terms of integrals over Ω_0 via the usual change of variables formula:

$$\int_{\Omega_e} f^e(x)\,dx = \int_{\Omega_0} f^0(X)|\det J(X)|\,dX.$$

We will generally assume that χ is *positively-oriented*, that is, $\det J$ is positive over all points within Ω_0. Under this assumption, we can convert quadrature rules over the reference domain to mapped quadrature rules:

$$\int_{\Omega_e} f^e(x)\,dx \approx \sum_{j=1}^{p} f^e(\chi(\xi_j)) \det J(\xi_j) w_j.$$

```
struct MappedRule{T,M} <: QuadratureRule where {T <: QuadratureRule}
    base_rule :: T                # Reference domain rule
    chi!      :: M                # Mapping
    J         :: Matrix{Float64}  # Space for Jacobian/LU
    ipiv      :: Vector{Int}      # Pivots for Jacobian LU
end

MappedRule(base_rule, chi!) =
    MappedRule(base_rule, chi!,
               zeros(quad_dim(base_rule), quad_dim(base_rule)),
               zeros(Int, quad_dim(base_rule)))

r2s_gradients!(q :: MappedRule, g) = r2s_gradients!(q.J, q.ipiv, g)

quad_npoints(q :: MappedRule) = quad_npoints(q.base_rule)
quad_dim(q :: MappedRule) = quad_dim(q.base_rule)

function quad_point(q :: MappedRule, i)
    x = quad_point(q.base_rule, i)
    q.chi!(x, q.J)
    r2s_factor!(q.J, q.ipiv)
    x
end
```

By default, `quad_point` is called before `quad_weight`, and so we do not need to re-evaluate the mapping and its Jacobian

```
function quad_weight(q :: MappedRule, i; map = false)
    if map quad_point(q, i) end
    quad_weight(q.base_rule, i) * r2s_det(q.J, q.ipiv)
end
```

Iso-parametric mapping

In some cases, the mapping function χ may be hand-crafted. Usually, though, we write χ as a combination of shape functions over the reference domain:

$$\chi(X) = \sum_{i=1}^{m} x_i N_i^e(X)$$

where x_i are given nodal locations. In this case, the Jacobian of the map is

$$\chi'(X) = \sum_{i=1}^{m} x_i N_i^{e\prime}(X).$$

When the same shape functions are used for defining the domain mapping and the interpolation of solution fields on the domain, we say we are using an *iso-parametric* mapping.

```
function isoparametric!(shapes, xnodal, x, J)
    shapes(x)
    mul!(x, xnodal, shapes.N,   1.0, 0.0)
    mul!(J, xnodal, shapes.dN', 1.0, 0.0)
    x
end
```

It is useful to also define an isoparametric quadrature rule

```
struct IsoMappedRule{T <: QuadratureRule,
                     S <: ShapeFuns,
                     M <: AbstractMatrix} <: QuadratureRule
    base_rule :: T                # Reference domain rule
    shapes    :: S                # Shapes
    xnodal    :: M                # Nodal points
    map_grads :: Bool             # Flag whether to map gradients
    J         :: Matrix{Float64}  # Space for Jacobian/LU
    ipiv      :: Vector{Int}      # Pivots for Jacobian LU
end

IsoMappedRule(base_rule, shapes, xnodal, map_grads=false) =
    IsoMappedRule(base_rule, shapes, xnodal, map_grads,
                  zeros(quad_dim(base_rule), quad_dim(base_rule)),
                  zeros(Int, quad_dim(base_rule)))

r2s_gradients!(q :: IsoMappedRule, g) = r2s_gradients!(q.J, q.ipiv, g)

quad_npoints(q :: IsoMappedRule) = quad_npoints(q.base_rule)
quad_dim(q :: IsoMappedRule) = quad_dim(q.base_rule)

function quad_point(q :: IsoMappedRule, i)
    x = quad_point(q.base_rule, i)
    isoparametric!(q.shapes, q.xnodal, x, q.J)
    r2s_factor!(q.J, q.ipiv)
    if q.map_grads r2s_gradients!(q, q.shapes) end
```

```
    x
end

function quad_weight(q :: IsoMappedRule, i; map = false)
    if map quad_point(q, i) end
    quad_weight(q.base_rule, i) * r2s_det(q.J, q.ipiv)
end
```

Mesh geometry

A mesh consists of an array of nodes locations $x_j \in \mathbb{R}^d$ and an element connectivity array with `elt[i,j]` giving the node number for the ith node of the jth element.

Each element represents a subset of $\Omega_e \subset \mathbb{R}^d$ that is the image of a reference domain $\Omega_0 \subset \mathbb{R}^d$ under a mapping

$$\chi(\xi) = \sum_{i=1}^{m} N_i^e(\xi) x_i$$

where $x_1, \ldots, x_m$ are the m element node positions. The functions N_i^e are nodal basis functions (or Lagrange basis functions, or cardinal functions) for an interpolation set $\xi_1, \ldots, \xi_m \in \Omega_0$; that is $N_i(\xi_j) = \delta_{ij}$. The reference domain nodes ξ_i are typically placed at corners or on edges of the reference domain, and their images are at corresponding locations in Ω_e.

When the same set of nodal basis functions (also called nodal shape functions in a finite element setting) are used both for defining the geometry and for approximating a PDE solution on Ω, we call this method of describing the geometry an *isoparametric* map.

We generally want our mappings describing the geometry to be *positively oriented*: that is, the map χ should be invertible and have positive Jacobian determinant over all of Ω_0. This puts some restrictions on the spatial positions of the nodes; for example, if the interpolation nodes appear in counterclockwise order in the reference domain Ω_0, then the corresponding spatial nodes in Ω_e should also appear in counterclockwise. order.

```
struct Mesh{T}
    shapes :: T             # Shape function interface
    X   :: Matrix{Float64}  # Node positions
    elt :: Matrix{Int}      # Connectivity
end

Mesh(shapes, numnp :: Integer, numelt :: Integer) =
    Mesh(shapes, zeros(dshapes(shapes), numnp),
         zeros(Int, nshapes(shapes), numelt))
```

Block meshers

One *can* allocate objects and then work out the node positions and element connectivity by hand (or with an external program). But in many cases, a simpler option is to programatically generate a mesh that covers a simple domain (e.g. a block) and then map the locations of the nodes. One can construct more complex meshes by combining this with a "tie" operation that merges the identity of nodes in the same location, but we will not bother with tied meshes for now.

The simplest mesher creates a 1D mesh on an interval $[a, b]$. We allow elements of order 1-3.

```
function mesh_create1d(numelt, shapes, a=0.0, b=1.0)
    nen   = nshapes(shapes)
    numnp = numelt * (nen-1) + 1
    mesh = Mesh(shapes, numnp, numelt)
    mesh.X[:] .= range(a, b, length=numnp)
    for j = 1:numelt
        mesh.elt[:,j] .= (j-1)*(nen-1) .+ (1:nen)
    end
    mesh
end
```

Things are more complicated in 2D, and we have distinct mesh generation routines for the different types of shape functions described in the `shapes` module. Each of these generates a mesh of the region $[0, 1]^2$ with `nex`-by-`ney` elements.

```
function mesh_block2d_P1(nex, ney)
    nx, ny = nex+1, ney+1
    mesh = Mesh(Shapes2dP1(), nx*ny, nex*ney)

    # Set up nodes (row-by-row, SW to NE)
    for iy = 1:ny
        for ix = 1:nx
            i = ix + (iy-1)*nx
            mesh.X[:,i] .= ( (ix-1)/(nx-1), (iy-1)/(ny-1) )
        end
    end

    # Set up element connectivity
    for iy = 1:ney
        for ix=1:nex
            i = ix + (iy-1)*nex
            i_sw = ix + (iy-1)*(nex+1)
```

```
            mesh.elt[:,i] •= (i_sw,
                              i_sw + 1,
                              i_sw + 1 + nex+1,
                              i_sw + nex+1)
        end
    end

    mesh
end

function mesh_block2d_P2(nex, ney)
    nx, ny = 2*nex+1, 2*ney+1
    mesh = Mesh(Shapes2dP2(), nx*ny, nex*ney)

    # Set up nodes (row-by-row, SW to NE)
    for iy = 1:ny
        for ix = 1:nx
            i = ix + (iy-1)*nx
            mesh.X[:,i] •= ( (ix-1)/(nx-1), (iy-1)/(ny-1) )
        end
    end

    # Set up element connectivity
    for iy = 1:ney
        for ix = 1:nex
            i = ix + (iy-1)*nex
            i_sw = 2*(ix-1) + 2*(iy-1)*nx + 1
            mesh.elt[:,i] •= (i_sw,
                              i_sw + 1,
                              i_sw + 2,
                              i_sw + 2 + nx,
                              i_sw + 2 + 2*nx,
                              i_sw + 1 + 2*nx,
                              i_sw +   + 2*nx,
                              i_sw +   + nx,
                              i_sw + 1 + nx)
        end
    end

    mesh
end
```

```
function mesh_block2d_S2(nex, ney)
    nx0, nx1 = 2*nex+1, nex+1  # Even/odd row sizes
    numnp = (ney+1)*nx0 + ney*nx1
    mesh = Mesh(Shapes2dS2(), numnp, nex*ney)

    # Set up nodes (row-by-row, SW to NE)
    for iy = 1:ney
        start = (iy-1)*(nx0+nx1)

        # Fill bottom row
        for ix = 1:nx0
            mesh.X[:,start+ix] .= ( (ix-1)/(nx0-1), (iy-1)/ney )
        end

        # Fill middle row
        start += nx0
        for ix = 1:nx1
            mesh.X[:,start+ix] .= ( (ix-1)/(nx1-1), (iy-0.5)/ney )
        end
    end

    # Fill top row
    start = ney*(nx0+nx1)
    for ix = 1:nx0
        mesh.X[:,start+ix] .= ( (ix-1)/(nx0-1), 1.0 )
    end

    # Set up element connectivity
    for iy = 1:ney
        for ix = 1:nex
            i = ix + (iy-1)*nex
            i_sw = 2*(ix-1) + (iy-1)*(nx0+nx1) + 1
            i_ww =   (ix-1) + (iy-1)*(nx0+nx1) + nx0 + 1
            i_nw = 2*(ix-1) + (iy-1)*(nx0+nx1) + nx0 + nx1 + 1
            mesh.elt[:,i] .= (i_sw,
                              i_sw + 1,
                              i_sw + 2,
                              i_ww + 1,
                              i_nw + 2,
                              i_nw + 1,
                              i_nw,
                              i_ww)
```

```
        end
    end

    mesh
end

function mesh_block2d_T1(nex, ney)
    nx, ny = nex+1, ney+1
    mesh = Mesh(Shapes2dT1(), nx*ny, 2*nex*ney)

    # Set up nodes (row-by-row, SW to NE)
    for iy = 1:ney+1
        for ix = 1:nex+1
            i = ix + (iy-1)*(nex+1)
            mesh.X[:,i] .= ( (ix-1)/(nx-1), (iy-1)/(ny-1) )
        end
    end

    # Set up element connectivity
    for iy = 1:ney
        for ix = 1:nex
            i = ix + (iy-1)*nex;
            i_sw = ix + (iy-1)*(nex+1);

            # Two triangles makes a square
            mesh.elt[:,2*i-1] .= (i_sw, i_sw + 1, i_sw + nex+1)
            mesh.elt[:,2*i  ] .= (i_sw + nex+1, i_sw + 1, i_sw + 1 + nex+1)
        end
    end

    mesh
end
```

Mesh output

For debugging, it is helpful to be able to print out all or part of the mesh geometry. We mostly care about this for looking at small meshes.

```
function mesh_print_nodes(mesh)
    @printf("\nNodal Positions:\n")
    @printf("   ID ")
```

```
    for j = 1:dshapes(mesh.shapes)
        @printf("     X%d", j)
    end
    @printf("\n")
    for i = 1:size(mesh.X,2)
        @printf("%3d : ", i)
        for j = 1:dshapes(mesh.shapes)
            @printf(" %6.2g", mesh.X[j,i])
        end
        @printf("\n")
    end
end

function mesh_print_elt(mesh)
    @printf("\nElement connectivity:\n")
    for i = 1:size(mesh.elt,2)
        @printf("% 3d :", i)
        for j = 1:size(mesh.elt,1)
            @printf("  % 3d", mesh.elt[j,i])
        end
        @printf("\n")
    end
end

function mesh_print(mesh)
    mesh_print_nodes(mesh)
    mesh_print_elt(mesh)
end
```

Assembly

Each element in a finite element discretization consists of

- A domain Ω_e for the eth element, and
- Local shape functions $N_1^e, \ldots, N_m^e$, which are often Lagrange functions for interpolation at some set of nodes in Ω_e.

Each local shape function on the domain Ω_e is the restriction of some global shape function on the whole domain Ω. That is, we have global shape functions

$$N_j(x) = \sum_{j=\iota(j',e)} N_{j'}^e(x),$$

where $\iota(j, e)$ denotes the mapping from the local shape function index for element e to the corresponding global shape function index. We only ever compute explicitly with the local functions N_j^e; the global functions are implicit.

Assembly is the process of reconstructing a quantity defined in terms of global shape functions from the contributions of the individual elements and their local shape functions. For example, to compute

$$F_i = \int_\Omega f(x) N_i(x)\, dx,$$

we rewrite the integral as

$$F_i = \sum_{i=\iota(i',e)} \int_{\Omega_e} f(x) N_{i'}^e(x)\, dx.$$

In code, this is separated into two pieces:

- Compute element contributions $\int_{\Omega_e} f(x) N_{i'}^e(x)\, dx$. This is the responsibility of the element implementation.
- Sum contributions into the global position i corresponding to the element-local index i'. This is managed by an assembly loop.

The concept of an "assembly loop" is central to finite element methods, but it is not unique to this setting. For example, circuit simulators similarly construct system matrices (conductance, capacitance, etc) via the contributions of circuit elements (resistors, capacitors, inductors, and so forth).

We have two types of assembly loops that we care about: those that involve pairs of shape functions and result in matrices, and those that explicitly involve only a single shape function and result in vectors.

Our assemblers all implement two methods

- `clear!(assembler)` – Clears things out
- `assemble_add!(assembler, econtrib, ids)` – add the element contribution `econtrib` at the locations indicated by `ids`. Any zero or negative indices are dropped.

Filtered loops

We will sometimes also want to discard some element contributions that correspond to interactions with shape functions associated with known boundary values (for example). We also handle this filtering work as part of our assembly process. Because we do this a lot, we define an iterator over valid `(i, id)` pairs, where `i` is the index in the element numbering system and `id` is the corresponding reduced index in the global system (with Dirichlet BC indices skipped).

```
struct IdIterator{T <: AbstractVector} ids :: T end

function Base.iterate(iter :: IdIterator, state=1)
    while state ⩽ length(iter.ids) && iter.ids[state] ⩽ 0
        state += 1
    end
    state > length(iter.ids) ? nothing : ((state, iter.ids[state]), state+1)
end
```

Vector assembly

```
clear!(v :: Vector) = (v[:] .= 0)

function assemble_add!(v :: Vector, evec, ids)
    for (i, id) in IdIterator(ids)
        v[id] += evec[i]
    end
end
```

Dense matrix assembly

```
clear!(A :: Matrix) = (A[:] .= 0)

function assemble_add!(A :: Matrix, emat, ids)
    for (j, idj) in IdIterator(ids)
        for (i, idi) in IdIterator(ids)
            A[idi,idj] += emat[i,j]
        end
    end
end
```

Coordinate form assembly

A coordinate form matrix (COO) is just a list of `(i,j,A[i,j])` tuples (which we will store in parallel arrays). We will follow the convention that entries with duplicate row/column indices are summed in the final matrix. We preallocate space for a certain number of entries, and keep a counter `nentries` for how much of that preallocation is used. We can reallocate if needed, but of course it is better not to do so.

```
mutable struct COOAssembler
    I :: Vector{Int}      # Row ids
    J :: Vector{Int}      # Column ids
    V :: Vector{Float64}  # Values
    nentries :: Int       # Number of entries saved
    m :: Int              # Rows in matrix
    n :: Int              # Columns in matrix
end

COOAssembler(nalloc :: Int, m, n=0) =
    COOAssembler(zeros(Int, nalloc), zeros(Int, nalloc),
                 zeros(nalloc), 0, m, n > 0 ? n : m)
```

Clearing the coordinate form assembler doesn't require filling any space with zeros – we just reset the `nentries` counter.

```
clear!(assembler :: COOAssembler) = (assembler.nentries = 0)
```

The `ensure_capacity!` function ensures that we have capacity for `ncontribs` more entries. If we do not have capacity in the pre-allocated space, we resize the arrays to either double the preallocated capacity or to accommodate the additional contributions, whichever is more.

```
function ensure_capacity!(assembler :: COOAssembler, ncontribs)
    nold = length(assembler.V)
    if assembler.nentries + ncontribs > nold
        nnew = max(assembler.nentries + ncontribs, 2*nold)
        resize!(assembler.I, nnew)
        resize!(assembler.J, nnew)
        resize!(assembler.V, nnew)
    end
    assembler
end
```

The `add_entry!` function adds a single entry, assuming that capacity has already been ensured.

```
function add_entry!(assembler :: COOAssembler, idi, idj, entry)
    assembler.nentries += 1
    assembler.I[assembler.nentries] = idi
    assembler.J[assembler.nentries] = idj
    assembler.V[assembler.nentries] = entry
end
```

Finally, the `assembler_add!` function ensures that we have enough capacity, and then adds all the entries from the element matrix.

```
function assemble_add!(assembler :: COOAssembler, emat, ids)
    ensure_capacity!(assembler, length(ids)^2)
    for (j, idj) in IdIterator(ids)
        for (i, idi) in IdIterator(ids)
            add_entry!(assembler, idi, idj, emat[i,j])
        end
    end
end
```

The Julia `SparseArrays` package has a built-in function already to convert a coordinate form matrix in parallel arrays into a compressed sparse column representation.

```
to_csc(a :: COOAssembler) =
    sparse(view(a.I, 1:a.nentries),
           view(a.J, 1:a.nentries),
           view(a.V, 1:a.nentries), a.m, a.n)
```

Compressed sparse column reassembly

The `CSCAssembler` keeps the data for assembling a compressed sparse column matrix. In addition to the storage for the matrix that we are assembling, we keep some auxiliary scratch storage for aggregating contributions to one column. This scratch storage is initialized to zeros, and should stay all zeros outside the `CSCAssembler` routines.

```
struct CSCAssembler{Tv,Ti}
    A :: SparseMatrixCSC{Tv,Ti}  # CSC storage structure
    Aj :: Vector{Tv}             # Scratch vector
end

CSCAssembler(A :: SparseMatrixCSC{Tv,Ti}) where {Tv,Ti} =
    CSCAssembler(A, zeros(Tv, A.m))

function clear!(a :: CSCAssembler)
    a.A.nzval[:] .= 0.0
    a.Aj[:] .= 0.0
end

function assemble_add!(a :: CSCAssembler, emat, ids)
```

```
    nids = length(ids)
    A, Aj = a.A, a.Aj
    for (j, idj) in IdIterator(ids)

        # Populate dense column scratch
        for (i, idi) in IdIterator(ids)
            Aj[idi] += emat[i,j]
        end

        # Extract from dense column
        k1, kn = A.colptr[ids[j]], A.colptr[ids[j]+1]-1
        for k = k1:kn
            A.nzval[k] += Aj[A.rowval[k]]
        end

        # Clear dense columns scratch
        for (i, idi) in IdIterator(ids)
            Aj[idi] = 0.0
        end
    end
end
```

Finite element mesh

My finite element mesh data structure is informed by lots of old Fortran codes, and mostly is a big pile of arrays. Specifically, we have the nodal arrays:

- `U`: Global array of solution values, *including* those that are determined by Dirichlet boundary conditions. Column j represents the unknowns at node j in the mesh.
- `F`: Global array of load values (right hand side evaluations of the forcing function in Poisson, for example; but Neumann boundary conditions can also contribute to `F`).
- `id`: Indices of solution values in a reduced solution vector. One column per node, with the same dimensions as `U` (and `F`), so that `ureduced[id[i,j]]` corresponds to `U[i,j]` when `id[i,j]` is nonnegative. The reduced solution vector contains only those variables that are not constrained a priori by boundary conditions; we mark the latter with negative entries in the `id` array.

In addition, we keep a mesh, an element type, and a quadrature rule. Note that for the moment, we are assuming only one element type per problem; we could have a separate array of element types (one per element) if we wanted more flexibility.

```
mutable struct FEMProblem{T,S}

    # Mesh data
    mesh :: Mesh

    # Element type (NB: can generalize with multiple types)
    etype :: T

    # Quadrature rule
    qrule :: S

    # Storage for fields
    U  :: Matrix{Float64}  # Global soln values (ndof-by-numnp)
    F  :: Matrix{Float64}  # Global force values (ndof-by-numnp)
    id :: Matrix{Int}      # Global to reduced ID map (ndof-by-nump)

    # Dimensions
    ndof :: Int
    nactive :: Int

end

function FEMProblem(mesh, etype, qrule, ndof)
    numnp = size(mesh.X,2)
    nactive = numnp * ndof
    U = zeros(ndof, numnp)
    F = zeros(ndof, numnp)
    id = zeros(Int, ndof, numnp)
    FEMProblem(mesh, etype, qrule, U, F, id, ndof, nactive)
end
```

Index setup

The `assign_ids!` function sets up the `id` array. On input, the `id` entries should be initialized so that boundary values are marked with negative numbers, and everything else is non-negative. On output, entries of `id` for variables not subject to essential boundary conditions will be assigned indices from 1 to `nactive` (and `nactive` will be updated appropriately).

```
function assign_ids!(fe :: FEMProblem)
    nactive = 0
    for j = 1:size(fe.mesh.X,2)
```

```
        for i = 1:fe.ndof
            if fe.id[i,j] ⩾ 0
                nactive += 1
                fe.id[i,j] = nactive
            end
        end
    end
    fe.nactive = nactive
end
```

Solution updates

The `update_U!` function applies an update to the internal state. That we compute `U[i,j] -= du_red[id[i,j]]` for `id[i,j] > 0`. If the update comes from $K^{-1}R$ where K is the reduced tangent and R the reduced residual, then applying the update will exactly solve the equation in the linear PDE case. However, we can also appy approximate updates (e.g. with an inexact solver for K), and the same framework works for Newton iterations for nonlinear problems.

```
function update_U!(fe :: FEMProblem, du_red)
    for j = 1:size(fe.mesh.X,2)
        for i = 1:fe.ndof
            if fe.id[i,j] > 0
                fe.U[i,j] -= du_red[fe.id[i,j]]
            end
        end
    end
end
```

Loads and BCs

The `set_load!` function and `set_dirichlet!` function update the forcing array `F` and the boundary data entries and `id` array markers in `U` and `id`, respectively.

```
function set_load!(fe :: FEMProblem, f :: Function)
    for i = 1:size(fe.mesh.X,2)
        f(view(fe.mesh.X,:,i), view(fe.F,:,i))
    end
end

function set_dirichlet!(fe :: FEMProblem, f :: Function)
```

```
    for i = 1:size(fe.mesh.X,2)
        f(view(fe.mesh.X,:,i), view(fe.id,:,i), view(fe.U,:,i))
    end
end
```

Assembly

The assembly loops iterate through the elements and produce a global residual and tangent stiffness based on the current solution state.

```
function assemble!(fe :: FEMProblem, R :: Vector, K)
    nlocal = nshapes(fe.mesh.shapes) * fe.ndof
    Re = zeros(nlocal)
    Ke = zeros(nlocal,nlocal)
    ids = zeros(Int, nlocal)
    clear!(R)
    clear!(K)
    for i = 1:size(fe.mesh.elt,2)
        ids[:] .= reshape(view(fe.id,:,view(fe.mesh.elt,:,i)),:)
        Re[:] .= 0.0
        Ke[:] .= 0.0
        element_dR!(fe, i, Re, Ke)
        assemble_add!(R, Re, ids)
        assemble_add!(K, Ke, ids)
    end
    R, K
end
```

Debugging printer

```
function fem_print(fe :: FEMProblem)
    @printf("\nNodal information:\n")
    @printf("       ID ")
    for j = 1:dshapes(fe.mesh.shapes)
        @printf("     X%d", j)
    end
    for j = 1:fe.ndof
        @printf("     U%d", j)
    end
```

```
    for j = 1:fe.ndof
        @printf("     F%d", j)
    end
    @printf("\n")
    for i = 1:size(fe.mesh.X,2)
        @printf("%3d : ", i)
        for j = 1:ndof
            @printf("% 3d ", fe.id[i])
        end
        for j = 1:dshapes(fe.mesh.shapes)
            @printf(" %6.2g", fe.mesh.X[j,i])
        end
        for j = 1:fe.ndof
            @printf(" % 6.2g", fe.U[j,i])
        end
        for j = 1:fe.ndof
            @printf(" % 6.2g", fe.F[j,i])
        end
        @printf("\n");
    end
    mesh_print_elt(fe.mesh)
end
```

Elements

Abstractly, for steady-state problems, we are finding $u(x) = \sum_j N_j(x) u_j$ via an equation

$$R(u, N_i) = 0$$

for all shape functions N_i that are not associated with essential boundary conditions. The element routines compute the contribution of one element to the residual R and to the tangent $\partial R / \partial u_j$.

Different types of equations demand different types of elements. Even for a single type of element, we may depend on things like PDE coefficients or choices of material parameters (as well as implementation details like the quadrature rule used for computing integrals). An `element_t` object type keeps all this information together. The `element_t` data type should be thought of as representing a *type* of element, and not one specific element; usually many elements share fundamentally the same data, differing only in which nodes they involve. In the language of design patterns, this is an example of a "flyweight" pattern.

The main interface for an element is a method

```
element_dR!(etype, fe, eltid, Re, Ke)
```

where `etype` is data for the element type, `fe` is a finite element mesh data structure, `eltid` is the index of the element in the mesh, and `Re` and `Ke` are pointers to storage for the element residual and tangent matrix contributions. Either `Re` or `Ke` can be `nothing`, indicating that we don't need that output.

Because we will usually use the element type associated with the finite element problem, we provide a convenience wrapper that fills in the `etype` argument to `element_dR!` with the problem `etype`.

```
element_dR!(fe :: FEMProblem, eltid, Re, Ke) =
    element_dR!(fe.etype, fe, eltid, Re, Ke)
```

Poisson element

Right now, we only have one element type, corresponding to a Poisson problem, written in weak form as

$$R(u, N_i) = \int_\Omega \left(\nabla N_i(x) \cdot \nabla u(x) - N_i(x) f(x)\right) \, d\Omega(x).$$

There are no PDE coefficients or other special parameters to keep track of for this element type. This is also a simple enough problem that it's easy to write dimension-independent code – no need for distinguishing 1D from 2D elements.

NB: We are not particularly careful in the current code about avoiding intermediate allocations.

```
struct PoissonElt end

function element_dR!(:: PoissonElt, fe :: FEMProblem, eltid, Re, Ke)
    s = fe.mesh.shapes
    eltj = view(fe.mesh.elt,:,eltid)
    X = view(fe.mesh.X,:,eltj)
    U = view(fe.U,1,eltj)
    F = view(fe.F,1,eltj)
    du = zeros(dshapes(s))
    for (x,wt) in IsoMappedRule(fe.qrule, s, X, true)
        fx =  F' * s.N
        mul!(du, s.dN, U)
        mul!(Re, s.dN', du,   wt, 1.0)
        mul!(Re, s.N,   fx,  -wt, 1.0)
        mul!(Ke, s.dN', s.dN, wt, 1.0)
```

```
    end
end
```