Simple Finite Element Method in C

David Bindel (program), Andrew Appel

Table of contents

Introduction

This document is automatically extracted from the sources found at https://github.com/VeriNum/simple_cfem. The code is written as a pedagogical example, and is coded in C to make it a reasonable target for verification with the Verified Software Toolchain.

The correctness proof is described here.

The rest of this page is the C program.

Memory allocation

The C `malloc`/`calloc` may return `NULL` to indicate out-of-memory. We would rather have functions that, if they return at all, guarantee to have allocated what's requested. * Also, convenient to have specific-typed versions of `calloc` for allocating arrays.

```
void *surely_malloc(size_t); // Guarantees to allocate, if it returns at all
double *double_calloc(int n);  // Allocate and zero an array of n doubles
int *int_calloc(int n);     // Allocate and zero an array of n ints

void double_clear(double *p, int n);    // Set an array of n doubles to zeros
```

Vector and matrix conveniences

To represent a dense matrix, we define a structure `densemat_t` consisting of dimensions followed by a data array.

```
typedef struct densemat_t {
    int m,n;  // rows, columns
    double data[0];  // Start of data array
} *densemat_t;

// Create and free
densemat_t densemat_malloc(int m, int n);
void densemat_free(densemat_t dm);

// Clear storage
void densematn_clear(double* data, int m, int n);
void densemat_clear(densemat_t dm);

// Print array (assumes column major order)
void densematn_print(double* data, int m, int n);
void densemat_print(densemat_t data);

// Cholesky factorization and solve (uses only upper triangular)
void densematn_cfactor(double* A, int n);
void densematn_csolve(double* R, double* x, int n);
void densemat_cfactor(densemat_t A);
void densemat_csolve(densemat_t R, double* x);

// LU factorization and solve
void densematn_lufactor(int* ipiv, double* A, int n);
void densematn_lusolve(int* ipiv, double* A, double* x, int n);
void densematn_lusolveT(int* ipiv, double* A, double* x, int n);
void densemat_lufactor(int* ipiv, densemat_t A);
void densemat_lusolve(int* ipiv, densemat_t A, double* x);
void densemat_lusolveT(int* ipiv, densemat_t A, double* x);

// Jacobian determinant from LU factorization
double densematn_lujac(int* ipiv, double* A, int n);
double densemat_lujac(int* ipiv, densemat_t A);

// Squared norm and norm computations
double data_norm2(double* data, int n);
```

```
double data_norm(double* data, int n);
double densemat_norm2(densemat_t dm);
double densemat_norm(densemat_t dm);
```

Accessor/setter functions for column-major indexing

To subscript a matrix, that is, to fetch and store individual matrix entries, we don't use C array indexing directly. Instead, we use these functions, which implement column-major indexing within a 1-dimensional array. Even to access a column vector, where you'd think C array indexing would be very natural, we use these 2-d functions. The reason is to simplify the proof. That is, we can specify and prove, once and for all, correctness of column-major indexing (and the connection to the matrix abstraction of the Rocq Mathematical components library), and then reason about the abstraction in C (that is, function call) using the abstraction in VST/MathComp/Rocq. You might worry that using a function call instead of directly subscripting the array would make the program slower, but it doesn't, since the C compiler can inline the function.

```
double densematn_get(double *data, int rows, int i, int j);
void densematn_set(double *data, int rows, int i, int j, double x);
void densematn_addto(double *data, int rows, int i, int j, double x);

double densemat_get(densemat_t dm, int i, int j);
void densemat_set(densemat_t dm, int i, int j, double x);
void densemat_addto(densemat_t dm, int i, int j, double x);
```

Memory management

- The `densemat_malloc` function allocates space for the head structure (which contains the first entry in the data array) along with space for the remainder of the `m*n` double precision numbers in the data array.

```
densemat_t densemat_malloc(int m, int n)
{
    densemat_t h = surely_malloc(sizeof(struct densemat_t) + (m*n)*sizeof(double));
    h->m=m;
    h->n=n;
    return h;
}
```

```
void densemat_free(densemat_t vm)
{
    free(vm);
}

void densematn_clear(double* data, int m, int n)
{
  double_clear(data,m*n);
}

void densemat_clear(densemat_t vm)
{
  densematn_clear(vm->data, vm->m, vm->n);
}

double densematn_get(double *data, int rows, int i, int j) {
  return data[i+j*rows];
}

void densematn_set(double *data, int rows, int i, int j, double x) {
  data[i+j*rows]= x;
}


void densematn_addto(double *data, int rows, int i, int j, double x) {
  data[i+j*rows] += x;
}

double densemat_get(densemat_t dm, int i, int j) {
  return densematn_get(dm->data,dm->m,i,j);
}

void densemat_set(densemat_t dm, int i, int j, double x) {
  densematn_set(dm->data,dm->m,i,j,x);
}


void densemat_addto(densemat_t dm, int i, int j, double x) {
  densematn_addto(dm->data,dm->m,i,j,x);
}
```

I/O

We provide a print routines as an aid to debugging. In order to make sure that modest size matrices can be printed on the screen in a digestible matter, we only print the first couple digits in each entry. Note that we assume column major layout throughout.

```
void densematn_print(double* data, int m, int n)
{
    printf("%d-by-%d\n", m, n);
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j)
      printf(" % 6.2g", densematn_get(data,m,i,j));
        printf("\n");
    }
}

void densemat_print(densemat_t vm)
{
    densematn_print(vm->data, vm->m, vm->n);
}
```

Cholesky factorization

For our finite element code, we will largely work with SPD matrices for which a Cholesky solve is appropriate. On input, we assume a column major representation in which the upper triangle represents the upper triangle of an SPD matrix; the lower triangle is ignored. On output, the upper triangle of the matrix argument is overwritten by the Cholesky factor. We will error out if we encounter a negative diagonal (in violation of the assumed positive definiteness).

We will not bother to show the wrapper around the `densematn` version.

```
void densematn_cfactor(double* A, int n)
{
  /* sdot method */
  int i,j,k;
  double s;
  for (j=0; j<n; j++) {
     for (i=0; i<j; i++) {
       s = densematn_get(A,n,i,j);
       for (k=0; k<i; k++)
     s = s - densematn_get(A,n,k,i)*densematn_get(A,n,k,j);
       densematn_set(A,n,i,j, s/densematn_get(A,n,i,i));
```

```
    }
    s = densematn_get(A,n,j,j);
    for (k=0; k<j; k++) {
        double rkj = densematn_get(A,n,k,j);
        s = s - rkj*rkj;
    }
    densematn_set(A,n,j,j, sqrt(s));
  }
}
```

The `densemat_csolve(R, x)` function assumes a Cholesky factor in the upper triangle of input argument `R`; the argument `x` is the right-hand side vector b on input, and the solution vector x on output.

```
void densematn_csolve(double* R, double* x, int n)
{
    // Forward substitution
    for (int i = 0; i < n; ++i) {
      double bi = densematn_get(x,n,i,0);
        for (int j = 0; j < i; ++j)
      bi -= densematn_get(R,n,j,i)*densematn_get(x,n,j,0);
    densematn_set(x,n,i,0, bi/densematn_get(R,n,i,i));
    }

    // Backward substitution
    for (int i = n; i > 0; --i) {
      // start loop at n to avoid negative indexes, in case we ever
      // want to use unsigned integers to increase the indexing range
      double yi = densematn_get(x,n,i-1,0);
        for (int j = i; j < n; ++j)
      yi -= densematn_get(R,n,i-1,j)*densematn_get(x,n,j,0);
    densematn_set(x,n,i-1,0, yi/densematn_get(R,n,i-1,i-1));
    }
}
```

LU factorization and solve

Even if the system matrices in a finite element code are SPD, the Jacobians that are used in mapped elements generally will not be. Therefore, we need a basic pivoted LU factorization along with the basic Cholesky.

The factorization routine overwrites `A` with the L and U factors, packed into the (strictly) lower and the upper triangular parts of A. The pivot vector is stored in `ipiv`, where `ipiv[i] = l` implies that rows i and l were swapped at step i of the elimination.

```
void densematn_lufactor(int* ipiv, double* A, int n)
{
    for (int j = 0; j < n; ++j) {

        // Find pivot row
        int ipivj = j;
        for (int i = j+1; i < n; ++i)
            if (fabs(A[i+n*j]) > fabs(A[ipivj+n*j]))
                ipivj = i;
        ipiv[j] = ipivj;

        // Apply row swap, if needed
        if (ipivj != j)
            for (int k = j; k < n; ++k) {
          double t = densematn_get(A,n,j,k);
          densematn_set(A,n,j,k, densematn_get(A,n,ipivj,k));
          densematn_set(A,n,ipivj,k, t);
            }

        // Compute multipliers
        double Ujj = densematn_get(A,n,j,j);
        for (int i = j+1; i < n; ++i)
      densematn_set(A,n,i,j, densematn_get(A,n,i,j)/Ujj);

        // Apply Schur complement update
        for (int k = j+1; k < n; ++k) {
      double Ujk = densematn_get(A,n,j,k);
            for (int i = j+1; i < n; ++i) {
          double Lij = densematn_get(A,n,i,j);
          densematn_addto(A,n,i,k, -Lij*Ujk);
            }
        }
    }
}
```

The `densemat_lusolve` function assumes that the factorization has already been computed. On input, `x` represents b; on output, `x` represents $x = A^{-1}b$.

```
void densematn_lusolve(int* ipiv, double* A, double* x, int n)
{
    // Apply P
    for (int i = 0; i < n; ++i)
        if (ipiv[i] ≠ i) {
            double t = x[i];
            x[i] = x[ipiv[i]];
            x[ipiv[i]] = t;
        }

    // Forward substitution
    for (int i = 0; i < n; ++i) {
        double bi = x[i];
        for (int j = 0; j < i; ++j)
      bi -= densematn_get(A,n,i,j)*x[j];
        x[i] = bi;
    }

    // Backward substitution
    for (int i = n; i ⩾ 0; --i) {
        double yi = x[i];
        for (int j = i+1; j < n; ++j)
      yi -= densematn_get(A,n,i,j)*x[j];
        x[i] = yi/densematn_get(A,n,i,i);
    }
}
```

The `densemat_lusolveT` variant solves a linear system $A^T x = b$ where $A^T = U^T L^T P$

```
void densematn_lusolveT(int* ipiv, double* A, double* x, int n)
{
    // Forward substitution (with U^T)
    for (int i = 0; i < n; ++i) {
        double bi = x[i];
        for (int j = 0; j < i; ++j)
      bi -= densematn_get(A,n,j,i)*x[j];
        x[i] = bi/A[i+i*n];
    }

    // Backward substitution (with L^T)
    for (int i = n; i ⩾ 0; --i) {
        double yi = x[i];
```

```
        for (int j = i+1; j < n; ++j)
      yi -= densematn_get(A,n,j,i)*x[j];
        x[i] = yi;
    }

    // Apply P^T
    for (int i = n-1; i ≥ 0; --i)
        if (ipiv[i] ≠ i) {
            double t = x[i];
            x[i] = x[ipiv[i]];
            x[ipiv[i]] = t;
        }
}
```

The Jacobian determinant can be computed from the product of the diagonals of U times the sign of the permutation matrix (given by the parity of the number of swaps in the factored permutation).

```
double densematn_lujac(int* ipiv, double* A, int n)
{
    double J = 1.0;
    int nswap = 0;
    for (int i = 0; i < n; ++i) {
        if (ipiv[i] ≠ i)
            ++nswap;
        J *= densematn_get(A,n,i,i);
    }
    return (nswap % 2 == 0) ? J : -J;
}
```

Norm computations

Just for checking on residuals and errors, it's convenient to have some functions for computing the squared Euclidean norm and the norm of a vector. We assume that things are sufficiently well scaled that we don't need to worry about over/underflow.

```
double data_norm2(double* data, int n)
{
    double result = 0.0;
    for (int j = 0; j < n; ++j) {
        double xj = data[j];
```

```
    result = fma(xj,xj,result);
    }
    return result;
}

double data_norm(double* data, int n)
{
    return sqrt(data_norm2(data, n));
}

double densemat_norm2(densemat_t vm)
{
    return data_norm2(vm->data, vm->m*vm->n);
}

double densemat_norm(densemat_t vm)
{
    return data_norm(vm->data, vm->m*vm->n);
}
```

Band matrix operations

We store symmetric band matrices using the LAPACK symmetric band format (see, e.g. `DSBTRF`). This is a packed storage format in which a symmetric matrix with `b` nonzero diagonals off th main diagonal in either direction is stored one diagonal at a time. That is, the dense matrix entry `A[i,j]` (column major) is stored in a packed array `P` of size `n`-by-`b+1` at location `P[j,d]`, where `d = j-i` is the diagonal number. The leading `d` entries of diagonal `d` are not used (but we don't try to eliminate them in the interest of keeping our indexing simple). Because we are interested in symmetric matrices, we only need to explicitly store the upper triangle (`d` $\geqslant$ `0`).

```
typedef struct bandmat_t {
    int m,b;  // rows, bands
    double data[0];  // Start of data array
} *bandmat_t;

// Allocate a new bandmat (and maybe populate from a dense matrix)
bandmat_t bandmat_malloc(int n, int b);
void bandmat_free(bandmat_t vm);
bandmat_t dense_to_band(densemat_t A, int bw);
```

```
// Clear

void bandmatn_clear(double* data, int m, int b);
void bandmat_clear(bandmat_t vm);


// Print a bandmat
void bandmat_print(bandmat_t PA);

// Frobenius norm-squared and norm
double bandmat_norm2(bandmat_t vm);
double bandmat_norm(bandmat_t vm);


// Cholesky and linear solve with Cholesky
void bandmat_factor(bandmat_t PA);
void bandmat_solve(bandmat_t PR, double* x);

// Getting, setting, adding to band matrices
double bandmatn_get(double *data, int rows, int i, int d);
void bandmatn_set(double *data, int rows, int i, int d, double x);
void bandmatn_addto(double *data, int rows, int i, int d, double x);

double bandmat_get(bandmat_t dm, int i, int d);
void bandmat_set(bandmat_t dm, int i, int d, double x);
void bandmat_addto(bandmat_t dm, int i, int d, double x);
```

Band matrix construction

```
// Allocate/free/clear a band matrix
bandmat_t bandmat_malloc(int n, int b)
{
  bandmat_t vm = surely_malloc(sizeof(struct bandmat_t) + (n*(b+1))*sizeof(double));
  vm->m=n; vm->b=b;
  return vm;
}

void bandmat_free(bandmat_t vm)
{
```

```
    free(vm);
}

void bandmatn_clear(double* data, int m, int b)
{
  memset(data, 0, (m*(b+1)) * sizeof(double));
}

void bandmat_clear(bandmat_t vm)
{
  bandmatn_clear(vm->data, vm->m, vm->b);
}



// Getting, setting, adding to band matrices
double bandmatn_get(double *data, int rows, int i, int d) {
  return data[i+d*rows];
}

void bandmatn_set(double *data, int rows, int i, int d, double x) {
  data[i+d*rows] = x;
}

void bandmatn_addto(double *data, int rows, int i, int d, double x) {
  data[i+d*rows] += x;
}

double bandmat_get(bandmat_t dm, int i, int d) {
  return bandmatn_get(dm->data, dm->m, i, d);
}

void bandmat_set(bandmat_t dm, int i, int d, double x) {
  bandmatn_set(dm->data, dm->m, i, d, x);
}

void bandmat_addto(bandmat_t dm, int i, int d, double x) {
  bandmatn_addto(dm->data, dm->m, i, d, x);
}


// Convert dense n-by-n A to band matrix P with bandwidth bw
```

```
bandmat_t dense_to_band(densemat_t A, int bw)
{
    int n = A->n;
    bandmat_t P = bandmat_malloc(n, bw);
    for (int d = 0; d <= bw; ++d)
        for (int j = d; j < n; ++j) {
            int i = j-d;
        bandmat_set(P,j,d, densemat_get(A,i,j));
        }
    return P;
}
```

Printing a band matrix

When printing a band matrix, we usually print just the structural nonzeros. Unless the matrix is very small, trying to introduce spaces or dots for structural zeros usually just makes the output too big to fit on a screen; hence, we will *almost* just print the underlying `n`-by-`b+1` data array. The only difference is that we will not bother to print out the "don't care" values that are at the start of the superdiagonal representations (since they will be garbage unless we zeroed them out, and anyway – we don't care about them).

```
// Print band format array
void bandmat_print(bandmat_t PA)
{
    int n = PA->m, bw = PA->b;

    for (int i = 0; i < n; ++i) {
        for (int d = 0; d <= bw && d <= i; ++d)
      printf("  % 6.3g", bandmat_get(PA,i,d));
        printf("\n");
    }
}
```

Band Cholesky and triangular solves

When computing a Cholesky factorization of a band matrix, the Schur complement update step only affects elements that were already structural nonzeros. Hence, Cholesky factorization of a band matrix can be done purely within the band data structure. The algorithm is essentially identical to the ordinary Cholesky factorization, except with indexing appropriate to the packed data structure. As with the dense Cholesky implementation in `densemat_t`, we only ever

reference the upper triangle of the matrix, and we overwrite the input arrays (representing the upper triangle of a symmetric input) by the output (representing an upper triangular Cholesky factor). Also as with dense Cholesky, we will error out if we encounter a negative diagonal in a Schur complement (violating the assumption of positive definiteness).

```
void bandmat_factor(bandmat_t PA)
{
    int n = PA->m, bw=PA->b;


    for (int k = 0; k < n; ++k) {

        // Compute kth diagonal element
        assert(bandmat_get(PA,k,0) >= 0);
    bandmat_set(PA,k,0, sqrt(bandmat_get(PA,k,0)));

        // Scale across the row
        for (int j = k+1; j < n && j <= k+bw; ++j)
      bandmat_set(PA,j,j-k, bandmat_get(PA,j,j-k)/bandmat_get(PA,k,0));

        // Apply the Schur complement update
        for (int j = k+1; j < n && j <= k+bw; ++j)
            for (int i = k+1; i <= j; ++i)
          bandmat_addto(PA,j,j-i, - bandmat_get(PA,i,i-k)*bandmat_get(PA,j,j-k));
    }
}
```

The `bandmat_solve(PR, x)` routine uses a band Cholesky factor R of the matrix A to solve $Ax = b$. The `PR` input argument gives the Cholesky factor (as computed by `bandmat_cholesky`); on input the `x` argument should be set to the system right-hand side, and on output it will be the solution vector.

```
void bandmat_solve(bandmat_t PR, double* x)
{
    int n = PR->m, bw = PR->b;

    // Forward substitution
    for (int i = 0; i < n; ++i) {
        double bi = x[i];
        for (int dj = 1; dj <= bw && dj <= i; ++dj)
      bi -= bandmat_get(PR,i,dj)*x[i-dj];
        x[i] = bi/bandmat_get(PR,i,0);
```

```
    }

    // Backward substitution
    for (int i = n-1; i >= 0; --i) {
        double yi = x[i];
        for (int j = i+1; j <= i+bw && j < n; ++j)
      yi -= bandmat_get(PR,j,j-i)*x[j];
        x[i] = yi/bandmat_get(PR,i,0);
    }
}
```

Norm computations

```
double bandmat_norm2(bandmat_t vm)
{
  return data_norm2(vm->data, vm->m*(vm->b+1));
}

double bandmat_norm(bandmat_t vm)
{
  return data_norm(vm->data, vm->m*(vm->b+1));
}
```

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 matris with d columns for a d-dimensional reference domain). Our shape function implementations all have the following interface, where the output arguments `N` and `dN` are used to store the shape function values and derivative values. If a `NULL` is given for either of these output arguments, we just skip that part of the computation.

The function returns the number of shape functions it computes.

```
typedef int (*shapes_t)(double*, double*, double*);
```

Our 1D shape functions are Lagrange polynomials for equally-spaced nodes in the interval $[-1, 1]$. We only go up to cubic polynomials ($p = 3$), as high-order polynomial interpolation through equally-spaced points is poorly behaved. When finite element codes implement very high order elements, they usually use a non-equispaced mesh (e.g. Gauss-Lobatto-Legendre nodes) that are better behaved for interpolation.

```
int shapes1dP1(double* N, double* dN, double* x);
int shapes1dP2(double* N, double* dN, double* x);
int shapes1dP3(double* N, double* dN, double* x);
```

The 2D P1 and P2 shape functions are tensor products of 1D P1 and P2 elements. The nodes are ordered counterclockwise, starting with the bottom left corner of the square. Thus, the P1 element has the reference domain $[-1, 1]^2$ and nodal points at the corners:

```
3 -- 2
|    |
0 -- 1
```

while for the P2 element, we include the mid-side nodes and one node in the middle (listed last):

```
6 -- 5 -- 4
|         |
7    8    3
|         |
0 -- 1 -- 2
```

The S2 element (part of the "serendipity family") is identical to the P2 element except that it does not include the final node in the middle.

```
int shapes2dP1(double* N, double* dN, double* x);
int shapes2dP2(double* N, double* dN, double* x);
int shapes2dS2(double* N, double* dN, double* x);
```

Finally, we define shape functions for a triangle with the reference domain with corners at $(0, 0)$, $(0, 1)$, and $(1, 0)$, listed in that order.

```
2
| \
0--1
```

```
int shapes2dT1(double* N, double* dN, double* x);
```

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.

```
typedef struct mesh_t {

    // Mesh storage
    double* X;  // Node locations (d-by-numnp)
    int* elt;   // Element connectivity array (nen-by-numelt)

    // Dimensions
    int d;       // Spatial dimension of problem
    int numnp;   // Number of nodal points
    int nen;     // Number of element nodes
    int numelt;  // Number of elements

    // Shape function
    int (*shape)(double* N, double* dN, double* x);

} *mesh_t;
```

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.

```
mesh_t mesh_malloc(int d, int numnp, int nen, int numelt);
void mesh_free(mesh_t mesh);
```

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

```
mesh_t mesh_create1d(int numelt, int degree, double a, double b);
```

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.

```
mesh_t mesh_block2d_P1(int nex, int ney);
mesh_t mesh_block2d_P2(int nex, int ney);
mesh_t mesh_block2d_S2(int nex, int ney);
mesh_t mesh_block2d_T1(int nex, int ney);
```

Given a mesh and a point in a reference geometry (given by an element identifier `eltid` and coordinates `xref` in the element's reference domain), we would like to be able to compute spatial quantities (the shape functions, their spatial derivatives, and the Jacobian of the reference to spatial map). The Jacobian matrix is in LU-factored form.

```
void mesh_to_spatial(mesh_t mesh, int eltid, double* xref,
                     double* x, int* ipiv, double* J,
                     double* N, double* dN);
```

We frequently are interested just in the mapped point location, shape functions and mapped derivatives, and the Jacobian determinant. So we provide a convenience wrapper around `mesh_to_spatial` for this case.

```
double mesh_shapes(mesh_t mesh, int eltid, double* x,
                   double* N, double* dN);
```

For debugging, it is helpful to be able to print out all or part of the mesh geometry.

```c
void mesh_print_nodes(mesh_t mesh);
void mesh_print_elt(mesh_t mesh);
void mesh_print(mesh_t mesh);
```

Memory management

```c
mesh_t mesh_malloc(int d, int numnp, int nen, int numelt)
{
    mesh_t mesh = surely_malloc(sizeof(struct mesh_t));
    mesh->d      = d;
    mesh->numnp  = numnp;
    mesh->nen    = nen;
    mesh->numelt = numelt;
    mesh->X      = double_calloc(d   * numnp);
    mesh->elt    = int_calloc(nen * numelt);
    mesh->shape  = NULL;
    return mesh;
}

void mesh_free(mesh_t mesh)
{
    free(mesh->elt);
    free(mesh->X);
    free(mesh);
}
```

Meshes in 1D

The simplest mesher creates a 1D mesh on an interval $[a, b]$. Elements are ordered from left to right. We allow elements of order 1-3.

```c
mesh_t mesh_create1d(int numelt, int degree, double a, double b)
{
    int numnp = numelt * degree + 1;
    int nen = degree + 1;
    mesh_t mesh = mesh_malloc(1, numnp, nen, numelt);

    if      (degree == 1) mesh->shape = shapes1dP1;
    else if (degree == 2) mesh->shape = shapes1dP2;
```

```
    else if (degree == 3) mesh->shape = shapes1dP3;
    else assert(0);

    // Set up equispaced mesh of points
    double* X = mesh->X;
    for (int i = 0; i < numnp; ++i)
        X[i] = (i*b + (numnp-i-1)*a)/(numnp-1);

    // Set up element connectivity
    int* elt = mesh->elt;
    for (int j = 0; j < numelt; ++j)
        for (int i = 0; i < nen; ++i)
            elt[i+j*nen] = i+j*(nen-1);

    return mesh;
}
```

Quad meshes in 2D

All the 2D quad meshers produce meshes of `nex` by `ney` elements, ordered in row-major order starting in the southwest and proceeding to the northeast. The nodes are listed going counterclockwise around the element (except possibly the last node in the P2 case).

We start with the P1 case, which is the simplest (only corner nodes).

```
mesh_t mesh_block2d_P1(int nex, int ney)
{
    int nx = nex+1, ny = ney+1;
    mesh_t mesh = mesh_malloc(2, nx*ny, 4, nex*ney);
    mesh->shape = shapes2dP1;

    // Set up nodes (row-by-row, SW to NE)
    for (int iy = 0; iy < ney+1; ++iy)
        for (int ix = 0; ix < nex+1; ++ix) {
            int i = ix + iy*(nex+1);
            mesh->X[2*i+0] = ((double) ix)/(nx-1);
            mesh->X[2*i+1] = ((double) iy)/(ny-1);
        }

    // Set up element connectivity
    for (int iy = 0; iy < ney; ++iy)
        for (int ix = 0; ix < nex; ++ix) {
```

```
            int i = ix + iy*nex;
            int i_sw = ix + iy*(nex+1);
            mesh->elt[4*i+0] = i_sw;
            mesh->elt[4*i+1] = i_sw + 1;
            mesh->elt[4*i+2] = i_sw + 1 + nex+1;
            mesh->elt[4*i+3] = i_sw + nex+1;
        }

    return mesh;
}
```

For P2 elements, each element involves three consecutive rows and columns of the logical array of nodes. This at least remains mostly straightforward.

```
mesh_t mesh_block2d_P2(int nex, int ney)
{
    int nx = 2*nex+1, ny = 2*ney+1;
    mesh_t mesh = mesh_malloc(2, nx*ny, 9, nex*ney);
    mesh->shape = shapes2dP2;

    // Set up nodes (row-by-row, SW to NE)
    for (int iy = 0; iy < ny; ++iy)
        for (int ix = 0; ix < nx; ++ix) {
            int i = ix + iy*nx;
            mesh->X[2*i+0] = ((double) ix)/(nx-1);
            mesh->X[2*i+1] = ((double) iy)/(ny-1);
        }

    // Set up element connectivity
    for (int iy = 0; iy < ney; ++iy)
        for (int ix = 0; ix < nex; ++ix) {
            int i = ix + iy*nex;
            int i_sw = (2*ix) + (2*iy)*nx;
            mesh->elt[9*i+0] = i_sw;
            mesh->elt[9*i+1] = i_sw + 1;
            mesh->elt[9*i+2] = i_sw + 2;
            mesh->elt[9*i+3] = i_sw + 2 + nx;
            mesh->elt[9*i+4] = i_sw + 2 + 2*nx;
            mesh->elt[9*i+5] = i_sw + 1 + 2*nx;
            mesh->elt[9*i+6] = i_sw +   + 2*nx;
            mesh->elt[9*i+7] = i_sw +   + nx;
            mesh->elt[9*i+8] = i_sw + 1 + nx;
```

```
        }

    return mesh;
}
```

The serendipity element block mesher is a little more complicated than P1 or P2, because we don't have a regular grid of mesh points (because we don't need mesh points in the middle of our elements.

```
mesh_t mesh_block2d_S2(int nex, int ney)
{
    int nx0 = 2*nex+1, nx1 = nex+1; // Even/odd row sizes
    int numnp = (ney+1)*nx0 + ney*nx1;
    mesh_t mesh = mesh_malloc(2, numnp, 8, nex*ney);
    mesh->shape = shapes2dS2;

    // Set up nodes (row-by-row, SW to NE)
    for (int iy = 0; iy < ney; ++iy) { // Element row index
        int start = iy*(nx0+nx1);

        // Fill bottom row
        for (int ix = 0; ix < nx0; ++ix) {
            mesh->X[2*(start+ix)+0] = ((double) ix)/(nx0-1);
            mesh->X[2*(start+ix)+1] = ((double) iy)/ney;
        }

        // Fill middle row
        start += nx0;
        for (int ix = 0; ix < nx1; ++ix) {
            mesh->X[2*(start+ix)+0] = ((double) ix)/(nx1-1);
            mesh->X[2*(start+ix)+1] = ((double) iy+0.5)/ney;
        }

        // Fill top row (may get overwritten by the same values shortly
        start += nx1;
        for (int ix = 0; ix < nx0; ++ix) {
            mesh->X[2*(start+ix)+0] = ((double) ix)/(nx0-1);
            mesh->X[2*(start+ix)+1] = ((double) iy+1.0)/ney;
        }
    }

    // Set up element connectivity
```

```
    for (int iy = 0; iy < ney; ++iy)
        for (int ix = 0; ix < nex; ++ix) {
            int i = ix + iy*nex;
            int i_sw = 2*ix + iy*(nx0+nx1);
            int i_ww =   ix + iy*(nx0+nx1) + nx0;
            int i_nw = 2*ix + iy*(nx0+nx1) + nx0+nx1;
            mesh->elt[8*i+0] = i_sw;
            mesh->elt[8*i+1] = i_sw + 1;
            mesh->elt[8*i+2] = i_sw + 2;
            mesh->elt[8*i+3] = i_ww + 1;
            mesh->elt[8*i+4] = i_nw + 2;
            mesh->elt[8*i+5] = i_nw + 1;
            mesh->elt[8*i+6] = i_nw;
            mesh->elt[8*i+7] = i_ww;
        }

    return mesh;
}
```

2D triangular meshes

The 2D linear triangle mesher is like the P1 mesher, but each quad is comprised of two triangles with a common edge going from the southeast to the northwest edge of the quad.

```
mesh_t mesh_block2d_T1(int nex, int ney)
{
    int nx = nex+1, ny = ney+1;
    mesh_t mesh = mesh_malloc(2, nx*ny, 3, 2*nex*ney);
    mesh->shape = shapes2dT1;

    // Set up nodes (row-by-row, SW to NE)
    for (int iy = 0; iy < ney+1; ++iy)
        for (int ix = 0; ix < nex+1; ++ix) {
            int i = ix + iy*(nex+1);
            mesh->X[2*i+0] = ((double) ix)/(nx-1);
            mesh->X[2*i+1] = ((double) iy)/(ny-1);
        }

    // Set up element connectivity
    for (int iy = 0; iy < ney; ++iy)
        for (int ix = 0; ix < nex; ++ix) {
```

```
            int i = ix + iy*nex;
            int i_sw = ix + iy*(nex+1);

            // Two triangles makes a square
            mesh->elt[6*i+0] = i_sw;
            mesh->elt[6*i+1] = i_sw + 1;
            mesh->elt[6*i+2] = i_sw + nex+1;
            mesh->elt[6*i+3] = i_sw + nex+1;
            mesh->elt[6*i+4] = i_sw + 1;
            mesh->elt[6*i+5] = i_sw + 1 + nex+1;
        }

    return mesh;
}
```

Reference to spatial mapping

```
void mesh_to_spatial(mesh_t mesh, int eltid, double* xref,
                     double* x, int* ipiv, double* J,
                     double* N, double* dN)
{
    int d = mesh->d;
    int* elt = mesh->elt + mesh->nen * eltid;
    double* X = mesh->X;

    // Get shape function
    int nshape = (*mesh->shape)(N, dN, xref);

    // Build x if requested
    if (x && N) {
        memset(x, 0, d * sizeof(double));
        for (int k = 0; k < nshape; ++k)
            for (int i = 0; i < d; ++i)
                x[i] += X[i+d*elt[k]] * N[k];
    }

    // Build and factor J and transform dN if requested
    if (ipiv && J && dN) {

        // Form J
```

```
        memset(J, 0, d * d * sizeof(double));
        for (int k = 0; k < nshape; ++k)
            for (int j = 0; j < d; ++j)
                for (int i = 0; i < d; ++i)
                    J[i+j*d] += X[i+d*elt[k]] * dN[k+j*nshape];

        // Factor
        densematn_lufactor(ipiv, J, d);

        // Transform shape derivatives to spatial coordinates
        for (int k = 0; k < nshape; ++k) {
            double dNk[3];
            for (int j = 0; j < d; ++j)
                dNk[j] = dN[k+j*nshape];
            densematn_lusolveT(ipiv, J, dNk, d);
            for (int j = 0; j < d; ++j)
                dN[k+j*nshape] = dNk[j];
        }
    }
}

double mesh_shapes(mesh_t mesh, int eltid, double* x,
                   double* N, double* dN)
{
    // Allocate space to make a 3D element work
    int ipiv[3];
    double J[9];
    double xout[3];
    int d = mesh->d;

    // Call mesh_to_spatial
    mesh_to_spatial(mesh, eltid, x, xout, ipiv, J, N, dN);
    memcpy(x, xout, d * sizeof(double));

    // If we asked for J, return the Jacobian
    return dN ? densematn_lujac(ipiv, J, d) : 0.0;
}
```

I/O routines

```
void mesh_print_nodes(mesh_t mesh)
{
    printf("\nNodal positions:\n");
    printf("   ID ");
    for (int j = 0; j < mesh→d; ++j)
        printf("     X%d", j);
    printf("\n");
    for (int i = 0; i < mesh→numnp; ++i) {
        printf("%3d : ", i);
        double* Xi = mesh→X + mesh→d*i;
        for (int j = 0; j < mesh→d; ++j)
            printf(" %6.2g", Xi[j]);
        printf("\n");
    }
}

void mesh_print_elt(mesh_t mesh)
{
    printf("\nElement connectivity:\n");
    for (int i = 0; i < mesh→numelt; ++i) {
        printf("% 3d :", i);
        for (int j = 0; j < mesh→nen; ++j)
            printf("  % 3d", mesh→elt[j + i*(mesh→nen)]);
        printf("\n");
    }
}

void mesh_print(mesh_t mesh)
{
    mesh_print_nodes(mesh);
    mesh_print_elt(mesh);
}
```

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.

Note that our code uses zero-based indexing (C-style) for indexing the quadrature nodes, even though the expression we wrote above uses the one-based indexing more common in presentations in the numerical methods literature.

```
double gauss_point(int i, int npts);
double gauss_weight(int i, int npts);
```

Product Gauss rules

A 2D tensor product Gauss rule for the domain $[-1, 1]^2$ involves a grid of quadrature points with coordinates given by 1D Gauss quadrature rules. We support rules with 1, 4, 9, or 16 points.

```
void gauss2d_point(double* xi, int i, int npts);
double gauss2d_weight(int i, int npts);
```

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.

```
void hughes_point(double* xi, int i, int npts);
double hughes_weight(int i, int npts);
```

Implementation

```
/*static*/ int gauss2d_npoint1d(int npts)
{
    switch (npts) {
    case  1: return 1;
    case  4: return 2;
    case  9: return 3;
    case 16: return 4;
    case 25: return 5;
    default:
        fprintf(stderr, "%d quadrature points unsupported by rule\n", npts);
        exit(-1);
    }
}

void gauss2d_point(double* xi, int i, int npts)
{
    int d = gauss2d_npoint1d(npts);
    int ix = i%d, iy = i/d;
    xi[0] = gauss_point(ix, d);
    xi[1] = gauss_point(iy, d);
}

double gauss2d_weight(int i, int npts)
{
    int d = gauss2d_npoint1d(npts);
    int ix = i%d, iy = i/d;
    return gauss_weight(ix, d) * gauss_weight(iy, d);
}
```

We only implement one triangle quadrature (the three-point Hughes rule).

```
void hughes_point(double* xi, int i, int npts)
{
    switch (i) {
    case 0:
```

```
        xi[0] = 0.5;
        xi[1] = 0.0;
        return;
    case 1:
        xi[0] = 0.5;
        xi[1] = 0.5;
        return;
    case 2:
        xi[0] = 0.0;
        xi[1] = 0.5;
        return;
    default:
        fprintf(stderr, "Quadrature node index out of bounds\n");
        exit(-1);
    }
}

double hughes_weight(int i, int npts)
{
    return 1.0/6.0;
}
```

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

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

In code, this is separated into two pieces:

- Compute element contributions $\int_{\Omega_e} f(x) N^e_{i'}(x)\, dx$. 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.

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.

Matrix assembler interface

There are several matrix formats that we might want to target as outputs for assembling a matrix; these include dense storage, banded storage, coordinate form, or CSR. Because we would like to re-use the same assembly loop logic with these different formats, we define an abstract `assemble_t` interface with two basic methods:

- `add(assembler, ematrix, ids, ne)` adds the `ne`-by-`ne` element matrix (`ematrix`) into the global structure referenced by the assembler. The `ids` array implements the map ι from local indices to global indices (i.e. `ids[ilocal] = iglobal`).
- `clear(assembler)` sets the matrix to zero, preserving the sparsity pattern ("graph").

```
// Interface for general assembler object (callback + context)
typedef struct assemble_data_t *assemble_data_t;

typedef struct assemble_t {
    assemble_data_t p;                          // Context
```

```
    void (*add)(assemble_data_t, double*, int*, int); // Add contribution
    void (*clear)(assemble_data_t); // set to zero
    double (*norm2)(assemble_data_t); // square of Frobenius norm
    void (*print)(assemble_data_t);
} *assemble_t;

// Convenience functions that call the assembler methods
void assemble_add(assemble_t assembler, double* emat, int* ids, int ne);
void assemble_clear(assemble_t assembler);
double assemble_norm2(assemble_t assembler);
void assemble_print(assemble_t assembler);
```

We currently only support two types of assemblers: dense and band. In all cases, we assume that the dimension `n` of the matrix is big enough (all active indices are less than `n`). For the band assembler, we do check to make sure there are no contributions that are outside the band (and error out if a contribution does live outside the expected band).

```
void init_assemble_dense(assemble_t assembler, densemat_t A);
void init_assemble_band(assemble_t assembler, bandmat_t b);
```

Vector assembly interface

We only really use one vector representation (a simple array), so there is no need for the same type of assembler abstraction for vectors that we have for matrices. The semantics of `assemble_vector` are similar to those of `assemble_add` in the matrix case, except now we add the element vector `ve` into the global vector `v`.

```
void assemble_vector(double* v, double* ve, int* ids, int ne);
```

Method dispatch

```
void assemble_add(assemble_t assembler, double* emat, int* ids, int ne)
{
    (*(assembler->add))(assembler->p, emat, ids, ne);
}

void assemble_clear (assemble_t assembler)
{
    (*(assembler->clear))(assembler->p);
```

```
}

double assemble_norm2 (assemble_t assembler)
{
   return (*(assembler->norm2))(assembler->p);
}

double assemble_norm (assemble_t assembler) {
  return sqrt(assemble_norm2(assembler));
}

void assemble_print (assemble_t assembler)
{
    (*(assembler->print))(assembler->p);
}
```

Setting up an assembler object just involves initializing the data pointer `p` and setting up the method table.

```
// Declare private implementations for the methods
/*static*/ void assemble_dense_add(assemble_data_t p, double* emat, int* ids, int ne);
/*static*/ void assemble_bandmat_add(assemble_data_t p, double* emat, int* ids, int ne);

// Initialize a dense assembler
void casted_densemat_clear(assemble_data_t p) {
  densemat_clear ((densemat_t)p);
}

double casted_densemat_norm2(assemble_data_t p) {
  return densemat_norm2 ((densemat_t)p);
}

void casted_densemat_print(assemble_data_t p) {
  densemat_print ((densemat_t)p);
}

void init_assemble_dense(assemble_t assembler, densemat_t A)
{
    assembler->p = (assemble_data_t)A;
    assembler->add = assemble_dense_add;
```

```
    assembler->clear = casted_densemat_clear;
    assembler->norm2 = casted_densemat_norm2;
    assembler->print = casted_densemat_print;
}

// Initialize a band assembler
void casted_bandmat_clear(assemble_data_t p) {
  bandmat_clear ((bandmat_t)p);
}

double casted_bandmat_norm2(assemble_data_t p) {
  return bandmat_norm2 ((bandmat_t)p);
}

void casted_bandmat_print(assemble_data_t p) {
  bandmat_print ((bandmat_t)p);
}

void init_assemble_band(assemble_t assembler, bandmat_t b)
{
    assembler->p = (assemble_data_t)b;
    assembler->add = assemble_bandmat_add;
    assembler->clear = casted_bandmat_clear;
    assembler->norm2 = casted_bandmat_norm2;
    assembler->print = casted_bandmat_print;
}
```

Matrix assembly loops

The assembly loops logically execute `A[iglobal, jglobal] += Ae[i, j]` for every local index pair `(i,j)`. We filter out the contributions where the global indices are negative (indicating that these contributions are not needed because of an essential boundary condition.

```
/*static*/ void assemble_dense_add(assemble_data_t p, double* emat, int* ids, int ne)
{
    densemat_t A = (densemat_t)p;

    for (int je = 0; je < ne; ++je) {
        int j = ids[je];
        for (int ie = 0; ie ⩽ je; ++ie) {
            int i = ids[ie];
```

```
            if (i ⩾ 0 && j ⩾ i)
          densemat_addto(A,i,j, emat[ie+ne*je]);
        }
    }
}

/*static*/ void assemble_bandmat_add(assemble_data_t p, double* emat, int* ids, int ne)
{
    bandmat_t P = (bandmat_t)p;
    int n = P->m;
    int b = P->b;

    for (int je = 0; je < ne; ++je) {
        int j = ids[je];
        for (int ie = 0; ie ⩽ je; ++ie) {
            int i = ids[ie];
            int d = j-i;
            if (j ⩾ 0 && d ⩾ 0) {
                assert(d ⩽ b);
        bandmat_addto(P,j,d, emat[ie+ne*je]);
            }
        }
    }
}
```

Vector assembly

```
void assemble_vector(double* v, double* ve, int* ids, int ne)
{
    for (int ie = 0; ie < ne; ++ie) {
        int i = ids[ie];
        if (i ⩾ 0)
            v[i] += ve[ie];
    }
}
```

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.

We also keep around a pointer to a mesh and an element type object. Note that for the moment, we are assuming only one element type per problem – we could have a separate array of element type pointer (one per element) if we wanted more flexibility.

```
typedef struct fem_t {

    // Mesh data
    struct mesh_t *mesh;

    // Element type (NB: can generalize with multiple types)
    struct element_t *etype;

    // Storage for fields
    double* U;  // Global array of solution values (ndof-by-numnp)
    double* F;  // Global array of forcing values (ndof-by-numnp)
    int* id;    // Global to reduced ID map (ndof-by-numnp)

    // Dimensions
    int ndof;    // Number of unknowns per nodal point (tested only with ndof = 1)
    int nactive; // Number of active dofs

} *fem_t;
```

Mesh operations

```
fem_t fem_malloc(mesh_t mesh, int ndof);
void fem_free(fem_t fe);
```

The `fem_assign_ids` function sets up the `id` array. On input, the `id` array in the mesh structure should be initialized so that boundary values are marked with negative numbers (and everything else non-negative). On output, entries of the `id` array for variables not subject to essential boundary conditions will be assigned indices from 0 to `nactive` (and `nactive` will be updated appropriately).

```
int fem_assign_ids(fem_t fe);
```

The `fem_update_U` function applies an update to the internal state. By update, we mean that `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 is the reduced residual, then applying the update will exactly solve the equation in the linear case. However, we can also apply approximate updates (e.g. with an inexact solver for K), and the same framework works for Newton iterations for nonlinear problems.

```
void fem_update_U(fem_t fe, double* du_red);
```

The `fem_set_load` function iterates through all nodes in the mesh, and for each node calls a callback function. The arguments to the callback are the node position (an input argument) and the node loading / right-hand side vector (an output argument).

```
void fem_set_load(fem_t fe, void (*f)(double* x, double* fx));
```

The assembly loops iterate through the elements and produce a global residual and tangent stiffness based on the current solution state. The residual and tangent matrix assembler are passed in by pointers; a `NULL` pointer means "do not assemble this".

```
void fem_assemble(fem_t fe, double* R, assemble_t Kassembler);
void fem_assemble_band(fem_t fe, double* R, bandmat_t K);
void fem_assemble_dense(fem_t fe, double* R, densemat_t K);
```

For debugging small problems, it is also useful to have a routine to print out all the mesh arrays.

```
void fem_print(fem_t fe);
```

Implementation

```
// Allocate mesh object
fem_t fem_malloc(mesh_t mesh, int ndof)
{
    int numnp = mesh->numnp;

    fem_t fe = surely_malloc(sizeof(struct fem_t));
    fe->mesh    = mesh;
    fe->etype   = NULL;
    fe->ndof    = ndof;
    fe->nactive = numnp * ndof;

    fe->U  = double_calloc(ndof * numnp);
    fe->F  = double_calloc(ndof * numnp);
    fe->id = int_calloc(ndof * numnp);

    return fe;
}

// Free mesh object
void fem_free(fem_t fe)
{
    free(fe->id);
    free(fe->F);
    free(fe->U);
    mesh_free(fe->mesh);
    free(fe);
}

// Initialize the id array and set nactive
int fem_assign_ids(fem_t fe)
{
    int numnp = fe->mesh->numnp;
    int ndof = fe->ndof;
    int* id = fe->id;
    int next_id = 0;
    for (int i = 0; i < numnp; ++i)
```

```
        for (int j = 0; j < ndof; ++j)
            if (id[j+i*ndof] >= 0)
                id[j+i*ndof] = next_id++;
    fe->nactive = next_id;
    return next_id;
}

// Decrement U by du_red
void fem_update_U(fem_t fe, double* du_red)
{
    double* U = fe->U;
    int* id   = fe->id;
    int ndof  = fe->ndof;
    int numnp = fe->mesh->numnp;
    for (int i = 0; i < numnp; ++i)
        for (int j = 0; j < ndof; ++j)
            if (id[j+i*ndof] >= 0)
                U[j+i*ndof] -= du_red[id[j+i*ndof]];
}

// Call the callback on each nodes (node position, force vector)
void fem_set_load(fem_t fe, void (*f)(double* x, double* fx))
{
    int d     = fe->mesh->d;
    int numnp = fe->mesh->numnp;
    int ndof  = fe->ndof;
    double* X = fe->mesh->X;
    double* F = fe->F;
    for (int i = 0; i < numnp; ++i)
        (*f)(X+i*d, F+i*ndof);
}

// Assemble global residual and tangent stiffness (general)
void fem_assemble(fem_t fe, double* R, assemble_t K)
{
    int numelt       = fe->mesh->numelt;
    int nen          = fe->mesh->nen;
    element_t etype = fe->etype;

    // Set up local storage for element contributions
    int* ids   =     int_calloc(nen);
    double* Re = R ? double_calloc(nen) : NULL;
```

```
    double* Ke = K ? double_calloc(nen*nen) : NULL;

    // Clear storage for assembly
    if (R) memset(R, 0, fe->nactive * sizeof(double));
    if (K) assemble_clear(K);

    // Assemble contributions
    for (int i = 0; i < numelt; ++i) {

        // Get element contributions
        element_dR(etype, fe, i, Re, Ke);

        // Figure out where to put them
        int* elt = fe->mesh->elt + i*nen;
        for (int j = 0; j < nen; ++j)
            ids[j] = fe->id[elt[j]];

        // Put them into the global vector/matrix
        if (R) assemble_vector(R, Re, ids, nen);
        if (K) assemble_add(K, Ke, ids, nen);

    }

    // Free local storage
    if (Ke) free(Ke);
    if (Re) free(Re);
    free(ids);
}

// Convenience function for assembling band matrix
void fem_assemble_band(fem_t fe, double* R, bandmat_t K)
{
    if (K) {
        struct assemble_t Kassembler;
        init_assemble_band(&Kassembler, K);
        fem_assemble(fe, R, &Kassembler);
    } else {
        fem_assemble(fe, R, NULL);
    }
}

// Convenience function for assembling dense matrix
```

```
void fem_assemble_dense(fem_t fe, double* R, densemat_t K)
{
    if (K) {
        struct assemble_t Kassembler;
        init_assemble_dense(&Kassembler, K);
        fem_assemble(fe, R, &Kassembler);
    } else {
        fem_assemble(fe, R, NULL);
    }
}

// Print mesh state
void fem_print(fem_t fe)
{
    printf("\nNodal information:\n");
    printf("       ID ");
    for (int j = 0; j < fe->mesh->d; ++j) printf("     X%d", j);
    for (int j = 0; j < fe->ndof; ++j)   printf("     U%d", j);
    for (int j = 0; j < fe->ndof; ++j)   printf("     F%d", j);
    printf("\n");
    for (int i = 0; i < fe->mesh->numnp; ++i) {
        printf("%3d : % 3d ", i, fe->id[i]);
        for (int j = 0; j < fe->mesh->d; ++j)
            printf(" %6.2g", fe->mesh->X[j+fe->mesh->d*i]);
        for (int j = 0; j < fe->ndof; ++j)
            printf(" % 6.2g", fe->U[j+fe->ndof*i]);
        for (int j = 0; j < fe->ndof; ++j)
            printf(" % 6.2g", fe->F[j+fe->ndof*i]);
        printf("\n");
    }
    mesh_print_elt(fe->mesh);
}
```

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

```
dR(p, fe, eltid, Re, Ke)
```

where `p` is context 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 null, indicating that we don't need that output.

We also provide a destructor method (`free`) for releasing resources used by the `element_t` instance.

```
// Element type interface
typedef struct element_t {
    void *p; // Context pointer
    void (*dR)(void* p, struct fem_t* fe, int eltid,
               double* Re, double* Ke);
    void (*free)(void* p);
} *element_t;

// Wrappers for calling the dR and free method
void element_dR(element_t e, struct fem_t* fe, int eltid,
                double* Re, double* Ke);
void element_free(element_t e);
```

Write now, we only have one element type, corresponding to a 1D 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 tyle.

```
element_t malloc_poisson1d_element(void);
element_t malloc_poisson2d_element(void);
```

Method dispatch

As usual for when we do OOP in C, we have dispatch functions that essentially trampoline a call to the appropriate function pointer in an element object's dispatch table.

```
// Call element dR method
void element_dR(element_t e, fem_t fe, int eltid,
                double* Re, double* Ke)
{
    (*(e->dR))(e->p, fe, eltid, Re, Ke);
}

// Call element free
void element_free(element_t e)
{
    (*(e->free))(e->p);
}
```

We write our Poisson interface to illustrate the general pattern, even though we could in principle simplify it (because we are not carrying around any element parameters in this case). The internal wiring is:

- Element type data is stored in a structure like `poisson1d_elt_t`.
- One field of the specific element is an `element_t` containing the methods table for the element.
- The data pointer in the `element_t` field points back to the containing struct (the `poisson1d_elt_t` in this case).

Externally, we always pass around `element_t` pointers. Internally, we always use the more specific `poisson1d_elt_t` from the `element_t` data pointer.

```
// Poisson element type data structure
typedef struct poisson_elt_t {
    // Material parameters, etc go here in more complex cases
    struct element_t e; // For dispatch table
} *poisson_elt_t;

// Declare methods for 1D and 2D Poisson element types
/*static*/ void poisson1d_elt_dR(void* p, fem_t fe, int eltid,
                                 double* Re, double* Ke);
/*static*/ void poisson2d_elt_dR(void* p, fem_t fe, int eltid,
                                 double* Re, double* Ke);
/*static*/ void simple_elt_free(void* p);
```

```
// Allocate a 1D Poisson element type
element_t malloc_poisson1d_element(void)
{
    poisson_elt_t le = (poisson_elt_t) surely_malloc(sizeof(struct poisson_elt_t));
    le->e.p = le;
    le->e.dR = poisson1d_elt_dR;
    le->e.free = simple_elt_free;
    return &(le->e);
}

// Allocate a 2D Poisson element type
element_t malloc_poisson2d_element(void)
{
    poisson_elt_t le = (poisson_elt_t) surely_malloc(sizeof(struct poisson_elt_t));
    le->e.p = le;
    le->e.dR = poisson2d_elt_dR;
    le->e.free = simple_elt_free;
    return &(le->e);
}

// Free a Poisson element type
/*static*/ void simple_elt_free(void* p)
{
    free(p);
}
```

1D Poisson element

The 1D Poisson element `dR` routine computes the local residual terms

$$R^e(u, N_i^e(x)) = \int_{\Omega_e} (\nabla N_i^e(x) \cdot \nabla u(x) - N_i^e(x) f(x)) \, d\Omega(x).$$

The functions $u(x)$ is represented on Ω_e in terms of the element shape functions

$$u(x) = \sum_i N_i^e(x) u_i$$

and similarly for $f(x)$. The tangent matrix has entries

$$\partial(R^e(u(x), N_i^e(x)))/\partial u_j = \int_{\Omega_e} \nabla N_i^e(x) \cdot \nabla N_j^e(x) \, d\Omega(x).$$

We organize the computation of the integrals (both for the residual vector and the tangent matrix) as an outer loop over quadrature nodes and inner loops over the shape function indices at the quadrature node.

We compute integrals using *mapped* quadrature: the locations of quadrature points in element reference domains are mapped to the element spatial domain, and the weights are multiplied by the Jacobian determinant for this computation.

```
/*static*/ void poisson1d_elt_dR(
    void* p,                    // Context pointer (not used)
    fem_t fe, int eltid,       // Mesh and element ID in mesh
    double* Re, double* Ke)    // Outputs: element residual and tangent
{
    int nen  = fe->mesh->nen;
    int ndof = fe->ndof;
    int degree = nen-1;
    int nquad = degree; // Would need one more for mass matrix...
    int* elt = fe->mesh->elt + eltid*nen;

    // Clear element storage
    if (Re) memset(Re, 0, nen*sizeof(double));
    if (Ke) memset(Ke, 0, nen*nen*sizeof(double));

    for (int k = 0; k < nquad; ++k) {

        // Get information about quadrature point (spatial)
        double N[4];  // Storage for shape functions
        double dN[4]; // Storage for shape derivatives
        double x  = gauss_point(k, nquad);
        double wt = gauss_weight(k, nquad);
        wt *= mesh_shapes(fe->mesh, eltid, &x, N, dN);

        // Add residual
        if (Re) {
            double du = 0.0;
            double fx = 0.0;
            double* U = fe->U;
            double* F = fe->F;
            for (int j = 0; j < nen; ++j) {
                du += dN[j]*U[ndof*elt[j]];
                fx += N[j]*F[ndof*elt[j]];
            }
            for (int i = 0; i < nen; ++i)
```

```
                Re[i] += (dN[i]*du - N[i]*fx) * wt;
        }

        // Add tangent stiffness
        if (Ke) {
            for (int j = 0; j < nen; ++j)
                for (int i = 0; i < nen; ++i)
                    Ke[i+j*nen] += dN[i]*dN[j] * wt;
        }
    }
}
```

Poisson elements in 2D

The 2D Poisson elements are very similar to the 1D case. As we wrote the formulas in a dimension-independent way in the documentation of the 1D case, we will not repeat ourselves here. The one thing that is a little different is that we will do a little more work to get an appropriate quadrature rule.

```
/*static*/ int get_quad2d(shapes_t shapefn,
                          void (**quad_pt)(double*, int, int),
                          double (**quad_wt)(int, int))
{
    if (shapefn == shapes2dP1) {
        *quad_pt = gauss2d_point;
        *quad_wt = gauss2d_weight;
        return 4;
    } else if (shapefn == shapes2dP2) {
        *quad_pt = gauss2d_point;
        *quad_wt = gauss2d_weight;
        return 9;
    } else if (shapefn == shapes2dS2) {
        *quad_pt = gauss2d_point;
        *quad_wt = gauss2d_weight;
        return 9;
    } else if (shapefn == shapes2dT1) {
        *quad_pt = hughes_point;
        *quad_wt = hughes_weight;
        return 3;
    } else
        assert(0);
```

```
    return 0; /* unreachable */
}

/*static*/ void poisson2d_elt_dR(
    void* p,                    // Context pointer (not used)
    fem_t fe, int eltid,       // Mesh and element ID in mesh
    double* Re, double* Ke)    // Outputs: element residual and tangent
{
    int nen  = fe->mesh->nen;
    int ndof = fe->ndof;
    void (*quad_pt)(double*, int, int);
    double (*quad_wt)(int, int);
    int nquad = get_quad2d(fe->mesh->shape, &quad_pt, &quad_wt);
    int* elt = fe->mesh->elt + eltid*nen;

    // Clear element storage
    if (Re) memset(Re, 0, nen*sizeof(double));
    if (Ke) memset(Ke, 0, nen*nen*sizeof(double));

    for (int k = 0; k < nquad; ++k) {

        // Get information about quadrature point (spatial)
        double N[4];    // Storage for shape functions
        double dN[4*2]; // Storage for shape derivatives
        double x[2];
        (*quad_pt)(x, k, nquad);
        double wt = (*quad_wt)(k, nquad);
        double J  = mesh_shapes(fe->mesh, eltid, x, N, dN);
        wt *= J;

        // Add residual
        if (Re) {
            double du[2] = {0.0, 0.0};
            double fx = 0.0;
            double* U = fe->U;
            double* F = fe->F;
            for (int j = 0; j < nen; ++j) {
                du[0] += U[ndof*elt[j]]*dN[j+0*nen];
                du[1] += U[ndof*elt[j]]*dN[j+1*nen];
                fx += N[j]*F[ndof*elt[j]];
            }
            for (int i = 0; i < nen; ++i)
```

```
                Re[i] += (dN[i+0*nen]*du[0]+dN[i+1*nen]*du[1] - N[i]*fx) * wt;
        }

        // Add tangent stiffness
        if (Ke) {
            for (int j = 0; j < nen; ++j)
                for (int i = 0; i < nen; ++i)
                    Ke[i+j*nen] += (dN[i]*dN[j]+dN[i+nen]*dN[j+nen]) * wt;
        }
    }
}
```