Simple 1D FEM in C

David Bindel

Table of contents

Vector and matrix conveniences

C does not make it particularly easy to work with matrices and vectors. Part of the difficulty is the lack of a convenient place to store size information. We work around this by defining a data structure (which we will refer to as a `vecmat_t`, though this type is never explicitly used in our code) consisting of dimension data followed by a data array. We generally pass the object around with a pointer to the start of the data (in standard C style), only backing up in memory to access size information when we need it.

```c
typedef struct vecmat_head_t {
    int m, n;       // Row and column counts
    double data[1]; // Start of data array
} vecmat_head_t;

// Get header information by backing up from data pointer
vecmat_head_t* vecmat(double* data);

// Create and free
double* malloc_vecmat(int m, int n);
void free_vecmat(double* data);

// Clear storage
void vecmat_clear(double* data);

// Print array (assumes column major order)
void vecmat_print(double* data);

// Cholesky factorization and solve (uses only upper triangular)
void vecmat_cfactor(double* A);
void vecmat_csolve(double* R, double* x);

// Squared norm and norm computations
double vecmat_norm2(double* data);
double vecmat_norm(double* data);
```

Memory management

We usually refer to a `vecmat_t` with a pointer to the (extended) `data` array whose start is declared in the `vecmat_head_t` structure. The `vecmat` function takes a double precision pointer to such a data field and backs up to get a pointer to the struct.

```c
vecmat_head_t* vecmat(double* data)
{
    vecmat_head_t dummy;
    void* p = (void*) data + ((void*) &dummy - (void*) dummy.data);
    return (vecmat_head_t*) p;
}
```

The `malloc_vecmat` 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. Because we want to be able to pass `vecmat_t` data into C functions that take plain pointers, we don't return the pointer to the head structure, but the pointer to the data field.

```c
double* malloc_vecmat(int m, int n)
{
    vecmat_head_t* h = malloc(sizeof(vecmat_head_t) + (m*n-1)*sizeof(double));
    h->m = m;
    h->n = n;
    return h->data;
}

void free_vecmat(double* data)
{
    free(vecmat(data));
}

void vecmat_clear(double* data)
{
    vecmat_head_t* vm = vecmat(data);
    int m = vm->m, n = vm->n;
    memset(data, 0, m*n * sizeof(double));
}
```

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.

```c
void vecmat_print(double* data)
{
    vecmat_head_t* vm = vecmat(data);
    int m = vm->m, n = vm->n;
    double* A = vm->data;
    printf("%d-by-%d\n", m, n);
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j)
            printf(" % 6.2g", A[i+j*m]);
        printf("\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).

```c
void vecmat_cfactor(double* A)
{
    vecmat_head_t* head = vecmat(A);
    int n = head->m;

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

        // Compute kth diagonal element
        double akk = A[k+n*k];
        assert(akk >= 0.0);
        double rkk = sqrt(akk);
        A[k+n*k] = rkk;

        // Scale across the row
        for (int j = k+1; j < n; ++j)
            A[k+n*j] /= rkk;

        // Apply the Schur complement update
        for (int j = k+1; j < n; ++j)
            for (int i = k+1; i <= j; ++i)
                A[i+j*n] -= A[k+i*n]*A[k+j*n];
    }
}
```

The `vecmat_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 vecmat_csolve(double* R, double* x)
{
    vecmat_head_t* head = vecmat(R);
    int n = head->m;

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

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

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 vecmat_norm2(double* data)
{
    vecmat_head_t* vm = vecmat(data);
    int m = vm->m, n = vm->n;
    double* A = vm->data;
    double result = 0.0;
    for (int j = 0; j < n; ++j)
        for (int i = 0; i < m; ++i) {
            double Aij = A[i+j*m];
            result += Aij*Aij;
        }
    return sqrt(result);
}

double vecmat_norm(double* data)
{
    return sqrt(vecmat_norm2(data));
}
```

Band matrix operations

We store symmetric band matrices using the LAPACK symmetric band format (see, e.g. `DS-BTRF`). 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 ⩾ 0`).

Because the storage format is essentially a dense `n`-by-`b+1` array, we will not introduce a totally new data structure for the band matrix; the `vecmat_t` storage container for dense matrices that we introduced before works well enough.

```
// Allocate a new bandmat (and maybe populate from a dense matrix)
double* malloc_bandmat(int n, int b);
double* dense_to_band(double* A, int n, int bw);

// Print a bandmat
void bandmat_print(double* PA);
```

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

Band matrix construction

```
// Allocate a band matrix
double* malloc_bandmat(int n, int b)
{
    return malloc_vecmat(n, b+1);
}

// Convert dense n-by-n A to band matrix P with bandwidth bw
double* dense_to_band(double* A, int n, int bw)
{
    double* P = malloc_bandmat(n, bw);
    for (int d = 0; d <= bw; ++d)
        for (int j = d; j < n; ++j) {
            int i = j-d;
            P[j+d*n] = A[i+j*n];
        }
    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(double* PA)
{
    vecmat_head_t* head = vecmat(PA);
    int n = head->m, bw = head->n-1;
    for (int i = 0; i < n; ++i) {
        for (int d = 0; d <= bw && d <= i; ++d)
            printf("  % 6.3g", PA[i+d*n]);
        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 `vecmat_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(double* PA)
{
    vecmat_head_t* head = vecmat(PA);
    int n = head->m, bw=head->n-1;

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

        // Compute kth diagonal element
        assert(PA[k] >= 0);
        PA[k] = sqrt(PA[k]);

        // Scale across the row
        for (int j = k+1; j < n && j <= k+bw; ++j)
            PA[j+n*(j-k)] /= PA[k];

        // Apply the Schur complement update
        for (int j = k+1; j < n && j <= k+bw; ++j)
            for (int i = k+1; i <= j; ++i)
                PA[j+n*(j-i)] -= PA[i+n*(i-k)]*PA[j+n*(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.

```c
void bandmat_solve(double* PR, double* x)
{
    vecmat_head_t* head = vecmat(PR);
    int n = head->m, bw = head->n-1;

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

    // 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 -= PR[j+(j-i)*n]*x[j];
        x[i] = yi/PR[i];
    }
}
```

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 $x_1, \dots, x_m$ 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 $\xi_1, \dots, \xi_m \in \Omega_0$; 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* malloc_mesh(int d, int numnp, int nen, int numelt);
void free_mesh(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);
```

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

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

Gaussian-Legendre quadrature rules

Gauss-Legendre quadrature rules (sometimes just called Gauss quadrature rules when the context is clear) approximate integrals with formulas of the form

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{j=1}^{p} f(\xi_{jp}) w_{jp},$$

where ξ_{jp} and w_{jp} are referred to as the quadrature nodes (or points) and weights, respectively. Gauss quadrature rules 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 $p = 10$, which are tabulated in many places.

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);
```

Assembly

Each element in a finite element discretization consists of

- A domain Ω_e for the eth element, and
- Local shape functions $N_1^e, \dots, N_m^e$, 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 $\int_{\Omega_e} f(x) N^e_{i'}(x)\,dx$. 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)` zeros out the matrix storage in preparation for assembly of a new matrix.

```
// Interface for general assembler object (callback + context)
typedef struct assemble_t {
    void* p;                                    // Context
    void (*add)(void*, double*, int*, int); // Add contribution
    void (*clear)(void*);                       // Clear
} 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);
```

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).

Both the dense and the band matrix expect pointers to data using our vecmat_t scheme, so we do not need to pass in explicit dimension arguments.

```
void init_assemble_dense(assemble_t* assembler, double* A);
void init_assemble_band(assemble_t* assembler, double* 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);
}
```

Setting up an assembler object just involves initializing the data pointer p and setting up the method table. Note that both the dense and band storage sit on top of our vecmat_t array manager, so we can use the same clear implementation in both cases.

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

```
// Initialize a dense assembler
void init_assemble_dense(assemble_t* assembler, double* A)
{
    assembler->p = A;
    assembler->add = assemble_dense_add;
    assembler->clear = assemble_vecmat_clear;
}

// Initialize a band assembler
void init_assemble_band(assemble_t* assembler, double* b)
{
    assembler->p = b;
    assembler->add = assemble_bandmat_add;
    assembler->clear = assemble_vecmat_clear;
}
```

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(void* p, double* emat, int* ids, int ne)
{
    vecmat_head_t* head = vecmat(p);
    double* A = head->data;
    int n = head->m;

    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)
                A[i+n*j] += emat[ie+ne*je];
        }
    }
}

static void assemble_bandmat_add(void* p, double* emat, int* ids, int ne)
{
    vecmat_head_t* head = vecmat(p);
    double* P = head->data;
    int n = head->m;
    int b = head->n-1;

    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);
                P[j+n*d] += emat[ie+ne*je];
            }
        }
    }
}
```

Clearing storage

```
static void assemble_vecmat_clear(void* p)
{
    vecmat_clear((double*) p);
}
```

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 positive. 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 (ndof = 1)
    int nactive; // Number of active dofs

} fem_t;
```

Mesh operations

```c
fem_t* malloc_fem(struct mesh_t* mesh, int ndof);
void free_fem(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).

```c
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.

```c
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).

```c
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".

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

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

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

Implementation

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

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

    fe->U  = (double*) calloc(ndof * numnp, sizeof(double));
    fe->F  = (double*) calloc(ndof * numnp, sizeof(double));
    fe->id = (int*)    calloc(ndof * numnp, sizeof(int));

    return fe;
}

// Free mesh object
void free_fem(fem_t* fe)
{
    free(fe->id);
    free(fe->F);
    free(fe->U);
    free_mesh(fe->mesh);
    free(fe);
}

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

```
            id[i] = 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    =     calloc(nen,     sizeof(int));
    double* Re = R ? calloc(nen,     sizeof(double)) : NULL;
    double* Ke = K ? calloc(nen*nen, sizeof(double)) : NULL;
```

```c
    // 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, double* K)
{
    if (K) {
        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, double* K)
{
    if (K) {
        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.

```c
// 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 free_element(element_t* e);
```

Write now, we only have one element type, corresponding to a 1D Poisson problem, written in weak form as

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

There are no PDE coefficients or other special parameters to keep track of for this element tyle.

```c
element_t* malloc_poisson_element();
```

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, struct fem_t* fe, int eltid,
                double* Re, double* Ke)
{
    (*(e->dR))(e->p, fe, eltid, Re, Ke);
}

// Call element free
void free_element(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 `poisson_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 `pois-son_elt_t` in this case).

Externally, we always pass around `element_t` pointers. Internally, we always use the more specific `poisson_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
    element_t e; // For dispatch table
} poisson_elt_t;

// Declare methods for Poisson element type
static void poisson_elt_dR(void* p, fem_t* fe, int eltid,
                           double* Re, double* Ke);
static void poisson_elt_free(void* p);

// Allocate a Poisson element type
```

```c
element_t* malloc_poisson_element()
{
    poisson_elt_t* le = (poisson_elt_t*) malloc(sizeof(poisson_elt_t));
    le->e.p = le;
    le->e.dR = poisson_elt_dR;
    le->e.free = poisson_elt_free;
    return &(le->e);
}

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

Mapped quadrature

We previously defined quadrature rules and element shape functions on a reference domain $[-1, 1]$; but our element subdomains are not all simple copies of this reference domain! Hence, we need a recipe for converting our rule on a reference domain with coordinates ξ into a rule on an element domain in real space with coordinates x.

A *mapped* quadrature rule uses a coordinate mapping $x = \chi(\xi)$ to convert integrals over the element domain to integrals over the reference domain. A standard choice for χ in the case of nodal shape functions is the *isoparametric map* that interpolates the nodal coordinates using the shape functions:

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

where x_i is the location of node i. We can also compute the Jacobian matrix (1-by-1 for 1D problems)

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

The change of variables formula for integration allows us to map integrals in our real element domain back to reference domain:

$$\int_{\Omega_e} f(x)\, dx = \int_{\Omega_0} f(x(\xi)) \det\left(\frac{\partial \chi}{\partial \xi}\right)\, d\xi.$$

Applying a quadrature rule with nodes ξ_i and weights w_i to the reference domain integral, we have

$$\int_{\Omega_e} f(x)\, dx \approx \sum_j f(x(\xi_j))\, \tilde{w}_j$$

where $\tilde{w}_j = \det(J(\xi_j))$ with $J(\xi)$ denoting the Jacobian $\partial\xi/\partial\xi$. Note that we assume that the Jacobian determinant of the mapping is positive.

Our weak forms generally involve derivatives of shape functions in x, but we so far only have the derivatives of the shape functions with respect to ξ. Converting derivatives in reference coordinates to spatial derivatives again involves the Jacobian determinant:

$$\frac{\partial N_i}{\partial x} = \frac{\partial N_i}{\partial \xi} J(\xi)^{-1}.$$

We note a (standard) abuse of notation here: the symbol N_i on the left hand side equation is technically the reference shape function N_i composed with the inverse coordinate mapping χ^{-1}.

Putting this all together, we want a function that at each quadrature node ξ_k will

- Compute shape functions and derivatives in the reference domain
- Use the isoparametric map to compute $x_k = \chi(\xi_k)$ and $J_k = \chi'(\xi_k)$.
- Use J_k to transform to spatial derivatives of the shape functions.
- Use J_k to compute the mapped quadrature weight.

The following local function does all these steps (in 1D).

```
static void set_qpoint1d(
    double* N,      // Shape functions at kth point
    double* dN,     // dN/dx at kth point
    double* xout,   // Location x of kth point
    double* wtout,  // Quadrature weight (with Jacobian)
    fem_t* fe,      // Finite element mesh structure
    int* elt,       // Connectivity for current element
    int k)          // Index of quadrature point
{
    int d      = fe->mesh->d;
    int nen    = fe->mesh->nen;
    int degree = nen-1;

    // Get reference domain quantities
    double xi = gauss_point(k, degree);
    double wt = gauss_weight(k, degree);
    (*fe->mesh->shape)(N, dN, &xi);

    // Map xi to spatial domain (and derivative dx/dxi)
    double x = 0.0;
    double dx_dxi = 0.0;
    double* X = fe->mesh->X;
    for (int i = 0; i < nen; ++i) {
        int ni = elt[i];
        x += N[i]*X[ni*d];
        dx_dxi += dN[i]*X[ni*d];
    }

    // Transform gradients and quadrature weight
    for (int i = 0; i < nen; ++i)
        dN[i] /= dx_dxi;
    wt *= dx_dxi;

    // Set output parameters
    *xout = x;
    *wtout = wt;
}
```

1D Poisson element

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

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

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.

```
static void poisson_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;
```

```c
    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, wt;
        set_qpoint1d(N, dN, &x, &wt, fe, fe->mesh->elt + eltid*nen, k);

        // 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;
        }
    }
}
```