C STRUCTURES AND SPARSE MATRICES

Daniel Shapero / shapero@uw.edu

OVERVIEW

- C structures are a way to aggregate data together into a single object.
- To introduce the concept, I'll show how to make a structure for a point in Euclidean space.
- For a more involved usage, I'll show how you would make a stack with and without structures.
- Finally, I'll show how to make a sparse matrix data type.

POINTS

- Suppose you're writing some code to do basic geometry.
- Some reasonable operations:
  - translation: $y = x + u$
  - scaling: $y = \alpha \cdot x$
  - rotation: $y = Q \cdot x$

SCALING

```c
void scale_point(double * x, double * y, double * z,
                 double x0, double y0, double z0,
                 double alpha)
{
    *x = alpha * x0;
    *y = alpha * y0;
    *z = alpha * z0;
}
```

TRANSLATION

```
void translate_point(double * x, double * y, double * z,
                     double x0, double y0, double z0,
                     double u0, double v0, double w0)
{
    *x = x0 + u0;
    *y = y0 + v0;
    *z = z0 + w0;
}
```

DO NOT WANT

- Writing out all the arguments is tedious.
- With structs, we can pack all the coordinates together:

```
struct point
{
  double x;
  double y;
  double z;
};
```

DECLARING AND ACCESSING FROM STRUCTS

- To declare a point, we have to use the keyword `struct` and the name `point`:

```
struct point P;
```

- The members of a struct are accessed using the `"."` operator:

```
struct point P;
double X = P.x;
P.y = 1729.0;
double length_square = P.x*P.x + P.y*P.y + P.z*P.z;
```

CREATING A STRUCT

- We can first declare a struct and then set all its members, or better yet we can initialize the whole struct at once:

```
struct point P = {.x = 1.0, .y = 0.0, .z = 0.0};
struct point Q = {.y = 1.0, .x = 0.0, .z = 0.0};
```

- The order doesn't matter if you use the member names.
- If the members are `const` then you have to do it this way.

GEOMETRY AGAIN

```
struct point scale(struct point P, double alpha)
{
    struct point Q =
      {.x = alpha * P.x, .y = alpha * P.y, .z = alpha * P.z};
    return Q;
}

struct point translate(struct point P, struct point U)
{
    struct point Q =
      {.x = P.x + U.x, .y = P.y + U.y, .z = P.z + U.z};
    return Q;
}
```

STRUCT POINTERS

- If a struct has lots of data, it's better to pass by reference than by value.
- When you only have a pointer to a struct, you have to dereference before accessing any members:

```
struct point * P = ...;
double X = (*P).x;
```

- The arrow operator -> is syntax sugar for dereference followed by member access:

```
struct point * P = ...;
double X = P->x;
```

GEOMETRY ONCE MORE

```
struct point scale(const struct point * P, double alpha)
{
    struct point Q =
      {.x = alpha * P->x, .y = alpha * P->y, .z = alpha * P->z};
    return Q;
}

struct point translate(const struct point * P,
                       const struct point * U)
{
    struct point Q =
      {.x = P->x + U->x, .y = P->y + U->y, .z = P->z + U->z};
    return Q;
}
```

LET'S LOOK AT REAL CODE

- From the GNU Triangulated Surface Library (GTL):

  ```
  struct _GtsPoint {
    GtsObject object;

    gdouble x, y, z;
  };
  ```

  in source file `gts.h`.
- The `GtsObject` member is for bookkeeping, but aside from that the implementation is the same as ours.

STACKS

- A stack is a data type for storing items so that the last item added can be retrieved quickly.
- "Last-in, first-out." Contrast this with queues, which are "first-in, first-out."
- Stacks support two operations:
  - push: add an element to the top of the stack
  - pop: take an element off the top of the stack

APPLICATIONS

- Puzzles: sudoku, crosswords
- Maze traversal
- Evaluating expressions in reverse-Polish notation

DYNAMIC ARRAYS

- Stacks can be implemented using an array which grows as need be if it reaches capacity.
- There are 3 variables to keep track of:
  - `int * data`: the array storing the stack contents
  - `size_t length`: the number of items on the stack
  - `size_t capacity`: the space allocated for the stack

POPPING FROM A STACK

```
int stack_pop(int ** data, size_t * length, size_t * capacity)
{
  assert(*length > 0);
  *length -= 1;
  return (*data)[*length];
}
```

RESIZING A STACK

```
void stack_resize(int ** data, size_t * length, size_t * capacity,
                  size_t new_capacity)
{
  *data = realloc(*data, new_capacity * sizeof(int));
  *capacity = new_capacity;
}
```

PUSHING TO A STACK

```c
void stack_push(int ** data, size_t * length, size_t * capacity,
                int item)
{
  if (*length == *capacity)
    stack_resize(data, length, capacity, 2 * (*capacity));

  (*data)[*length] = item;
  *length += 1;
}
```

DO NOT WANT

- That had lots of boilerplate.
- We always pass the `data`, `length`, `capacity` arguments together -- with structs, we can pack them all together.
- A stack is pretty simple. Imagine how much worse this would get if we had a complicated data type, like a tree.

STRUCTURE DEFINITION

- This is a sensible structure definition for a stack:

```
struct stack
{
  int * data;
  size_t length;
  size_t capacity;
};
```

- Note that the stack now contains a pointer to some memory. We have to allocate that memory to use the stack, and free it to avoid a memory leak.

STACKS ON STACKS ON STACKS

- Rather than manually set the stack members, we should write a convenience function to create a new stack.

```
struct stack make_stack(size_t capacity)
{
  int * data = calloc(capacity, sizeof(int));
  struct stack s = {.data = data,
                    .length = 0,
                    .capacity = capacity};
  return s;
}
```

CLEANING UP STRUCTURES

- But if the structure stores heap memory, it needs to be deallocated when we're done with it:

```
void stack_free(struct stack * s)
{
  if (s->data)
  {
      free(s->data);
      s->data = NULL;
  }

  s->length = 0;
  s->capacity = 0;
}
```

- Good practice: make sure you can call a destructor twice on the same object without nuking the whole program

STACK POP

```c
int stack_pop(struct stack * s)
{
    assert(s->length > 0);
    s->length -= 1;
    return s->data[s->length];
}
```

STACK PUSH

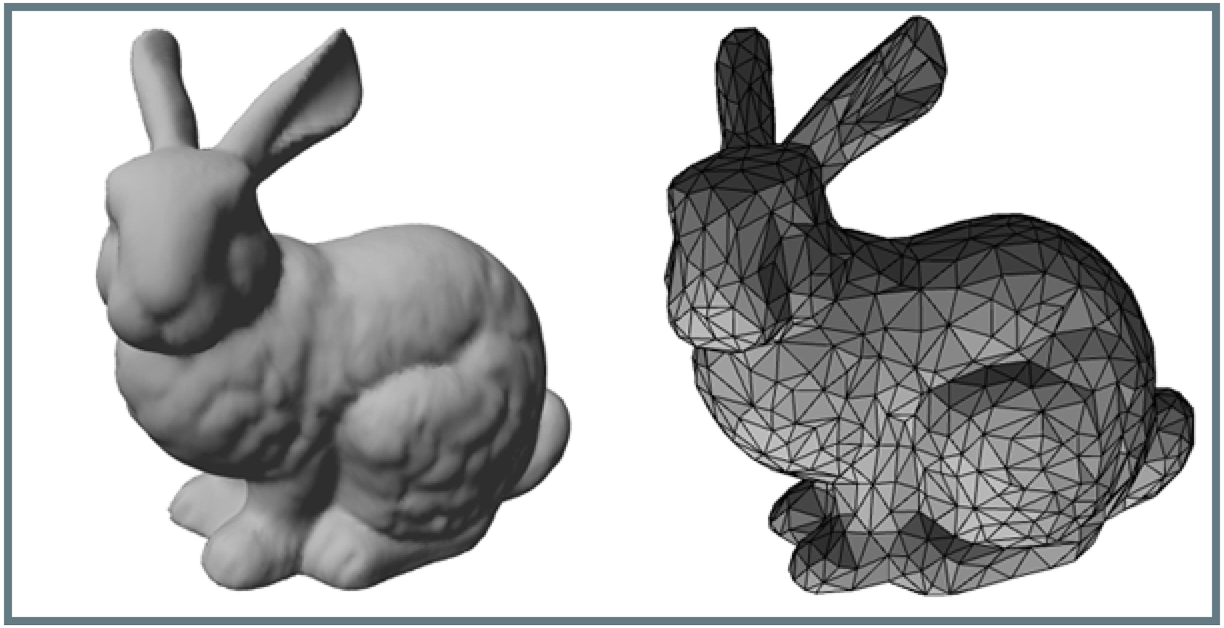
```
void stack_push(struct stack * s, int item)
{
    if (s->length == s->capacity)
    {
        s->capacity = max(2 * s->capacity, 1);
        s->data = realloc(s->data, sizeof(int) * s->capacity);
    }

    s->data[s->length] = item;
    s->length += 1;
}
```

APPLICATION: SPARSE MATRICES

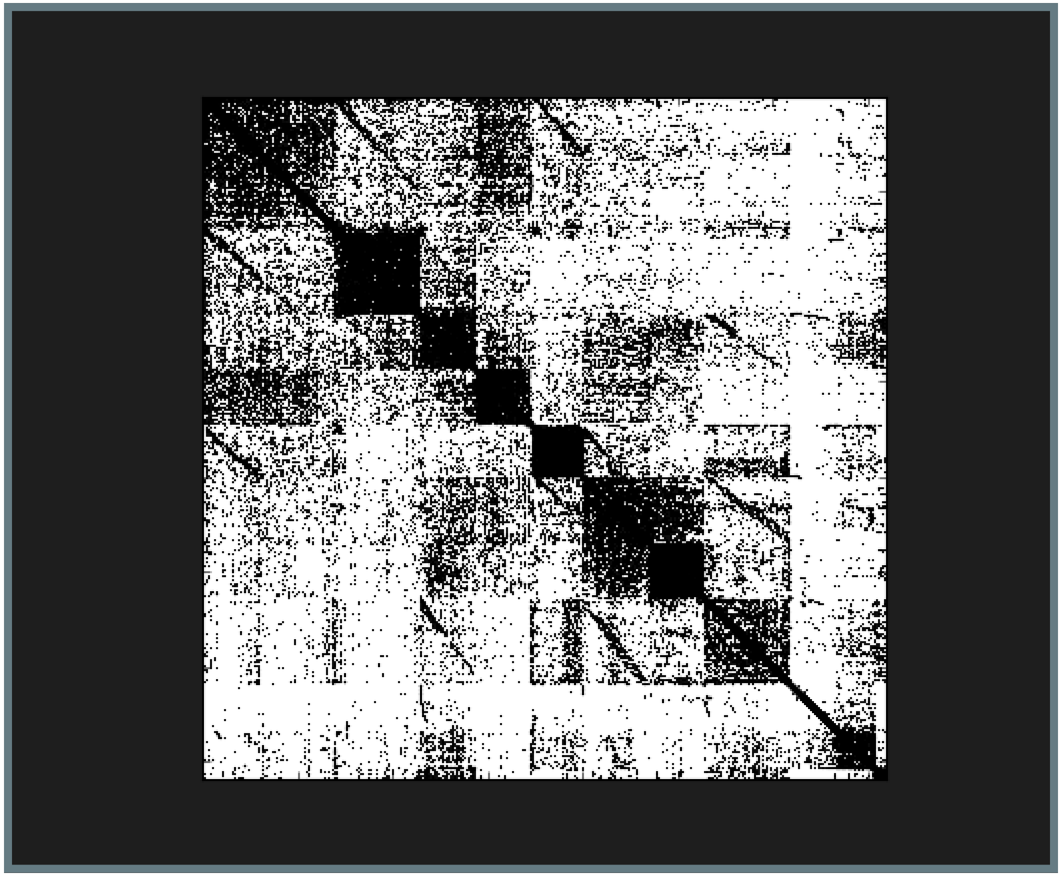
- So far, you've seen a lot of problems with tridiagonal matrices.
- Matrices where most of the entries are 0 are called *sparse*.
- Lots of problems give rise to sparse linear systems.

EXAMPLE: PDE DISCRETIZATIONS ON MESHES



The famous Stanford bunny

MATRIX OF THE BUNNY MESH



Generated using matplotlib spy

SPARSE MATRIX DATA STRUCTURES

- As an example, consider the matrix

$$A = \begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{bmatrix}$$

- How can we store only the non-zero entries?

THE COO FORMAT

- The simplest way to store a sparse matrix is using arrays for all the rows, columns and values:

```
rows: 0   0   1   1   1   2   2   2   2   3   3   4
cols: 0   3   0   1   3   0   2   3   4   2   3   4
vals: 1   2   3   4   5   6   7   8   9  10  11  12
```

- What would the corresponding C struct look like?

```
struct coo_matrix
{
    size_t num_rows, num_cols;
    size_t num_nonzero_entries;
    size_t * rows;
    size_t * columns;
    double * values;
};
```

MATRIX-VECTOR MULTIPLICATION

How do we compute $y = y + A \cdot x$ when A is stored in the COO format?

```
void coo_matvec(const struct coo_matrix * A,
                const double * x,
                double * y)
{
    for (size_t k = 0; k < A->num_nonzero_entries; ++k)
    {
        size_t i = A->rows[k];
        size_t j = A->columns[k];

        y[i] += A->values[k] * x[j];
    }
}
```

OTHER MATRIX OPERATIONS

- What else might we need to do with a matrix?
  - retrieve entry `(i, j)`
  - find out how many entries there are in row `i`
  - permute the entries
- The coordinate format requires O(nnz) time to do the first two operations. Can we do better than that?

SORTING THE MATRIX

- In a COO matrix, we have big runs of the same number:

```
rows: 0    0    1    1    1    2    2    2    2    3    3    4
```

- For an arbitrary matrix, we can always sort it by row.
- Rather than store this array, we can compress it.

THE CSR FORMAT

- We can do better using the run-length encoding of `rows`; call this array `offsets`.
- `offsets[i]` = beginning index of row `i` in the matrix
- Some other facts about the run-length encoding of `rows`:
  - `offsets[i] = 0`
  - `offsets[i+1]-offsets[i]` = # of entries in row `i`

AN EXAMPLE

The example matrix from before:

$$A = \begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{bmatrix}$$

The CSR format:

```
offsets: 0       2           5                9       11  12
columns: 0   3   0   1   3   0   2   3   4    2    3    4
values:  1   2   3   4   5   6   7   8   9   10   11   12
```

THE CSR FORMAT

- This is the most commonly used format in applications.
- Most operations are $O(d)$ where d is the degree of a row.
- Modifying pre-existing entries of the matrix can be done fast, but adding an entry that isn't already in the matrix structure could require re-allocating everything.

PERMUTATIONS

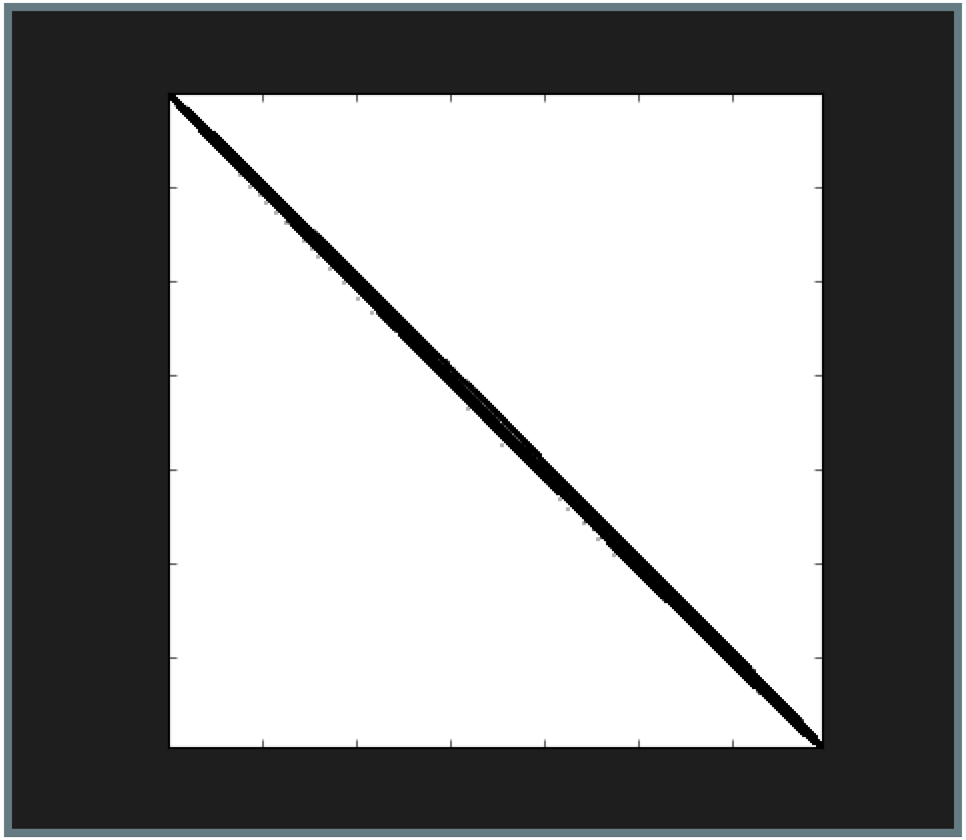
- Dense matrix factorizations require pivoting, i.e. permuting the entries, for numerical stability.
- There are even more uses for permutations of sparse matrices.
- We'll consider only one, the problem of *bandwidth reduction*.

BANDWIDTH

- Given a sparse matrix A, the *bandwidth* of A is
$$b = \max_i \max_{\{j : A_{ij} \neq 0\}} |j - i|.$$
- Can we find a permutation matrix P such that $P^* AP$ has lower bandwidth than A?

THE CUTHILL-MCKEE ORDERING

This is a permutation of the bunny matrix from before:



THE CUTHILL-MCKEE ORDERING

- Reducing the bandwidth makes multiplication 5-10% faster for this matrix. For some matrices, it can be >25%.
- Why would reordering make a difference if it's the same number of floating point operations?