P6 – Scientific Programming

Marcus Mohr
Jens Oeser

Geophysics Section
Department of Earth and Environmental Sciences
Ludwig-Maximilians-Universität München

SoSe 2021

Part #15

Data Structures

abstract data-types, trees, stacks, queues,

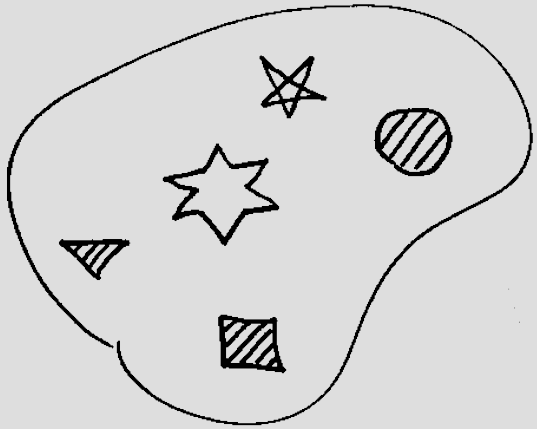
linked lists and containers

Data Structures

we all know / have encountered data-structures

8 × Brezen
12 × Weißwürscht
6 × Weizen
1 × süßer Senf

a shopping list



a set

Marcus Mohr
(Numerical Simulation)

Department of Earth and Environmental Sciences
Geophysics
Munich University
Theresienstr. 41
80333 Munich
Germany

Room: 411
Phone: +49 (89) 2180-4230
Fax: +49 (89) 2180-4205

E-Mail: marcus.mohr@geophysik.uni-muenchen.de
Internet: http://www.geophysik.uni-muenchen.de/~mohr

a data-base record

Data Structures (cont.)

Definition:

An organization of information, usually in memory, for better algorithm efficiency, such as queue, stack, linked list, heap, dictionary, and tree,

or conceptual unity, such as the name and address of a person. It may include redundant information, such as length of the list or number of nodes in a subtree.

From: Dictionary of Algorithms and Data Structures

Data Structures (cont.)

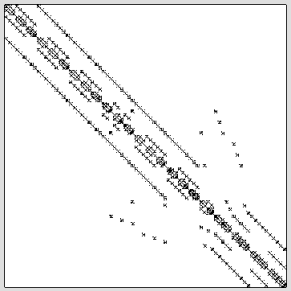
The main motivations for investigating/designing data structures are

- to find optimal data layouts
- that allow to efficiently manipulate/adminstrate data
- for certain applications
- to design ways to handle associated data concertedly

Object-oriented programming takes this to a limit by encapsulating data and methods for manipulating them into objects/classes.

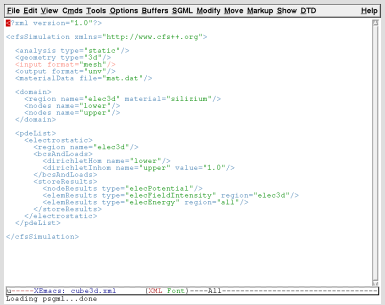
Scientific Programming

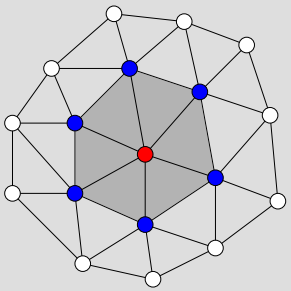
Specialised data structures are not only the realm of computer science, but also valuable for applications in computational science.



Sparse matrices require specialised storage schemes, like e.g. compressed row storage

Complex simulations require a significant amount of steering parameters. We may want to group these into a single data structure for handling.





Efficient book-keeping on neighbouring elements for a FEM grid is non-trivial.

Language Support

If one considers the concrete implementation of a data-structure in a certain programming language, one often speaks of an abstract data-type as opposed to the standard intrinsic types (integer, real, ...).

All modern languages (e.g. C, C++, Fortran90/95/2003) support the construction of abstract data-types using standard types as building blocks. These are often (e.g. in Fortran) denoted as derived types.

FORTRAN77, however, does not directly support the construction of derived types. (too old)

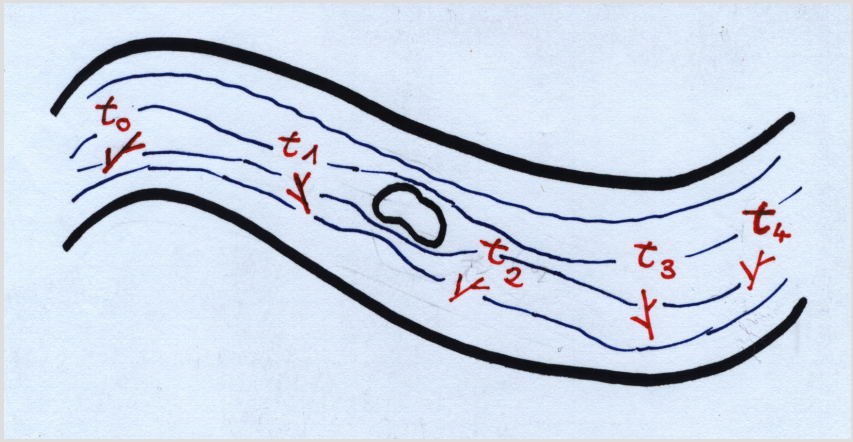
Principle Concepts Overview

type	comment	access
array	fixed number of data-items of same type are stored contiguously, supported directly in most languages	by index
list	dynamic storage of arbitrarily many data-items, each item contains a link to its successor	traversal
stack	dynamic storage of arbitrarily many data-items, last item stored must be accessed first (operations: push & pop)	LIFO
queue	dynamic storage of arbitrarily many data-items, first item stored must be accessed first (operations: enqueue & dequeue)	FIFO
tree	special structure for modelling hierarchies, many specialisations like e.g. balanced trees, octrees, ...	
heap	implements set structure, elements associated with keys for ordering ($\rightarrow$ priority queues)	

Application: Particle Tracing

One possibility to visualise the behaviour of a flow pattern (used e.g. in computational fluid dynamics) is particle tracing

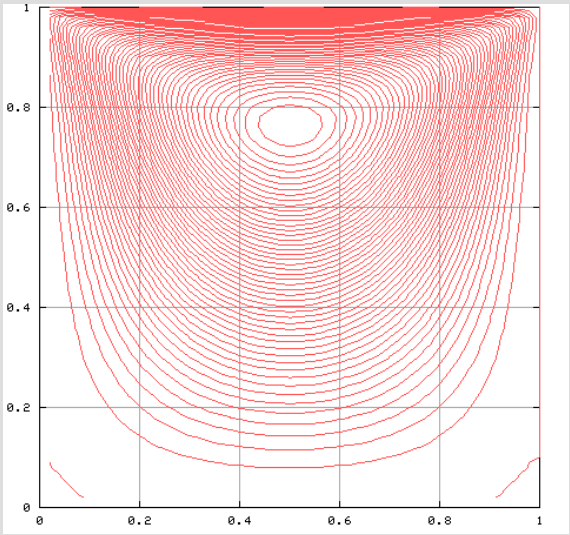
- at starting time insert particles into flow
- in each time step t_k compute current position of particles
- in intervals store current positions and visualise



"throw a stick into the river and watch it float"

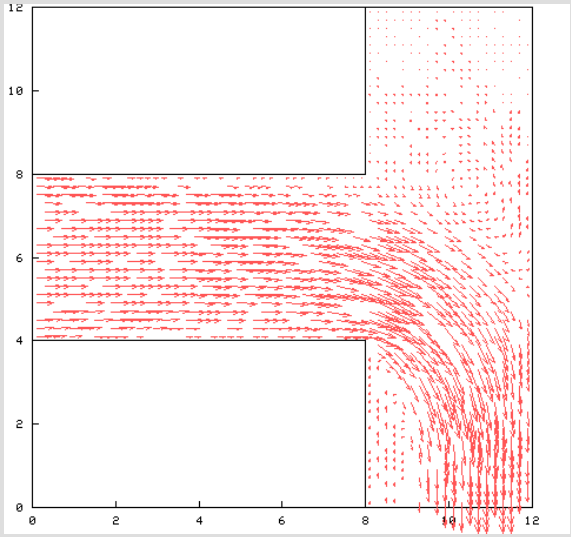
Application: Particle Tracing (cont.)

lid driven cavity



simple case:
number of particles
is constant

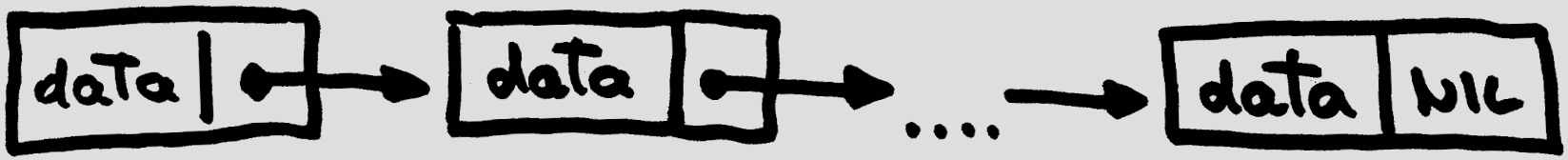
pipe junction



more complex:
number of particles
changes over time

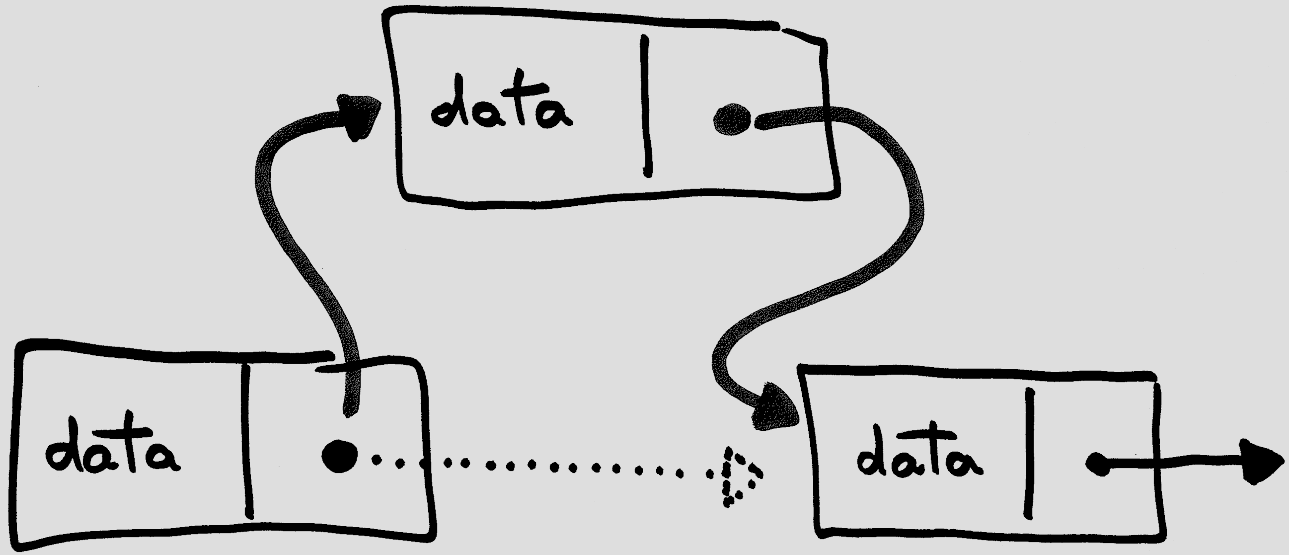
Singly Linked Lists

- A suitable data structure for storing a changing number of data-items with repeated insertions and deletions is a singly linked list.
- The costs for inserting and deleting one data item are constant.



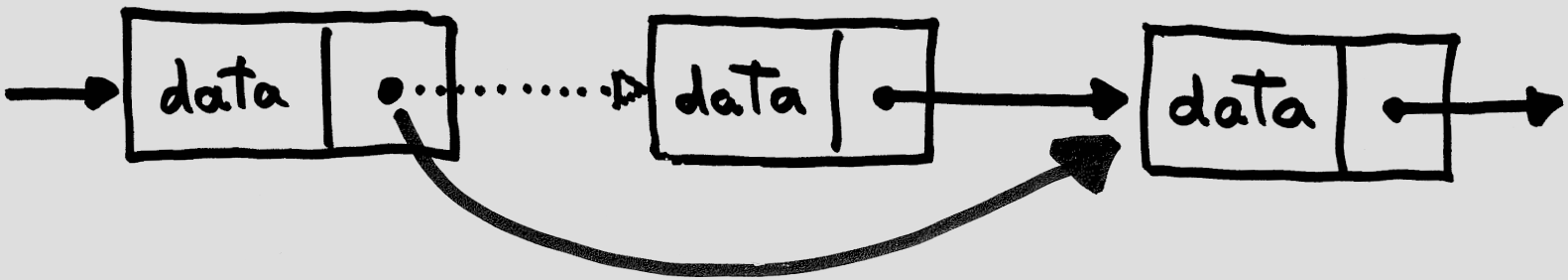
- Simplest form of a list. Each item contains a link to its successor.
- Last item needs a special NIL link to indicate end of list.

Linked List: Insertion



- split list at desired insertion point (after left neighbour of new item)
- insert link in left item to new item
- insert link in new item to right item

Linked List: Deletion



- split list in front of item to be deleted
- let predecessor of item point to its successor
- destroy item (if applicable)

Linked Lists in C

- In a language with dynamic memory allocation (such as C) we can in principle store an arbitrarily long list (within the limits of the available memory)
- A link in this case is simple the memory address of the next list item (stored in a pointer variable)
- Note, however, that this need not be the most efficient approach with respect to run-time (traversing the list means jumping around in memory)
- We will now perform a demo implementation of a linked list for particle tracing.

Example: Particle Tracing (1/8)

As a first step we define two data structures:

```
// datatype for a single particle
typedef struct party {
  double x, y;
  struct party* next;
} particle_t;

// datatype for all particles (as linked list)
typedef struct {
  particle_t* firstParticle;
  int length;
} particleLine_t;
```

Why do we need party?

Example: Particle Tracing (2/8)

Now a function for generating a new particle:

```
// place a new particle in the flow at given coordinates
particle_t* newParticle( double x, double y ) {

  // allocate new particle
  particle_t* aux = malloc( sizeof( particle_t ) );

  // set particle coordinates
  aux->x = x;
  aux->y = y;

  // no successor, yet
  aux->next = NULL;

  return aux;
}
```

Example: Particle Tracing (3/8)

And a function for deleting a particle:

```
// delete an existing particle
void delParticle( particle_t* particle ) {
  if( particle != NULL ) {
    free( particle );
  }
}
```

Example: Particle Tracing (4/8)

```
// The function will insert a particle after predecessor
particle_t* addToList( particle_t* predecessor, particle_t* particle,
                       particleLine_t* pLine ) {

  // first particle needs special treatment
  if( predecessor == NULL ) {
    pLine->firstParticle = particle;
    particle->next = NULL;
  }
  else {
    // get link (address of) successor and set it in the new particle
    particle->next = predecessor->next;

    // insert link to (address of) new particle in predecessor
    predecessor->next = particle;
  }

  // list got longer
  pLine->length++;

  return particle;
}
```

Example: Particle Tracing (5/8)

```
// remove current particle from list (need predecessor, as we cannot
// go backwards in _singly_ linked list)
void removeFromList( particle_t* predecessor, particle_t* current,
                     particleLine_t* pLine ) {

  // skip over current particle (removing it from the list)
  // if current particle is first in list it needs special treatment
  if( predecessor == NULL ) {
    pLine->firstParticle = current->next;
  }
  else {
    predecessor->next = current->next;
  }

  // delete particle object
  delParticle( current );

  // list got shorter
  pLine->length--;
}
```

Example: Particle Tracing (6/8)

Working with the list: list traversal to output particle info

```
void showParticles( particleLine_t* pLine ) {

  particle_t* current;
  current = pLine->firstParticle;

  printf( "# ---------------------------------\n" );
  printf( "# no. particles in line = %d\n", pLine->length );

  while( current != NULL ) {
    printf( "particle position: ( %4.2f , %4.2f )\n",
            current->x, current->y );
    current = current->next;
  }
  printf( "# ---------------------------------\n" );
}
```

Now let us write a test-driver:

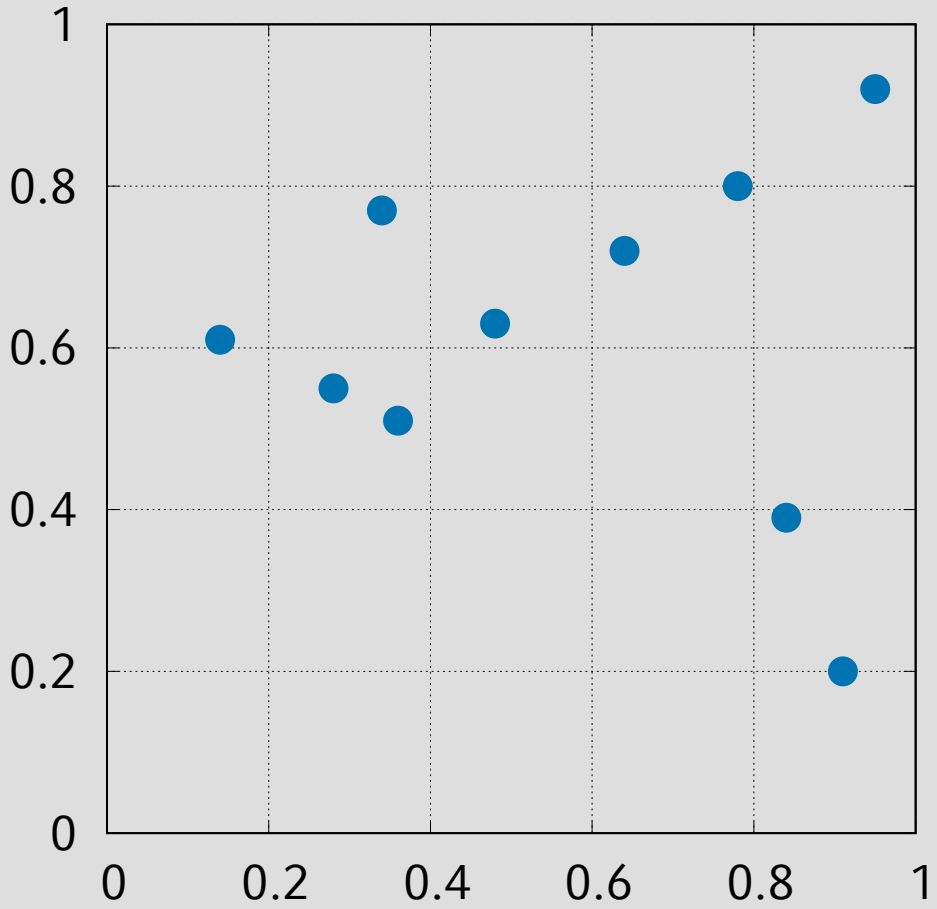
```
int main( void ) {

  particle_t* single = NULL;
  particle_t* current = NULL;
  particleLine_t pLine = { NULL, 0 };

  // generate some particles to fill the list initially
  int num = 10;
  current = NULL;
  for( int k = 0; k < num; k++ ) {
    double xPos = (double)rand() / (double)RAND_MAX;
    double yPos = (double)rand() / (double)RAND_MAX;
    single = newParticle( xPos, yPos );
    current = addToList( current, single, &pLine );
  }

  // show what we got
  showParticles( &pLine );
```

```
// now remove all particles from list with x > 0.5
current = pLine.firstParticle;
particle_t* previous = NULL;
particle_t* next = NULL;

while( current != NULL ) {
  next = current->next;
  if( current->x > 0.5 ) {
    removeFromList( previous, current, &pLine );
  }
  else {
    previous = current;
  }
  current = next;
}
printf( "\n\n" );
showParticles( &pLine );

// add another particle with coords (0.8,0.2) as
// last but one particle
current = pLine.firstParticle;
for( int k = 1; k < pLine.length - 1; k++ ) {
  current = current->next;
}
addToList( current, newParticle( 0.8, 0.2 ), &pLine );
printf( "\n\n" );
showParticles( &pLine );
```

Example: Particle Tracing (8/8)

we add 10 random points:

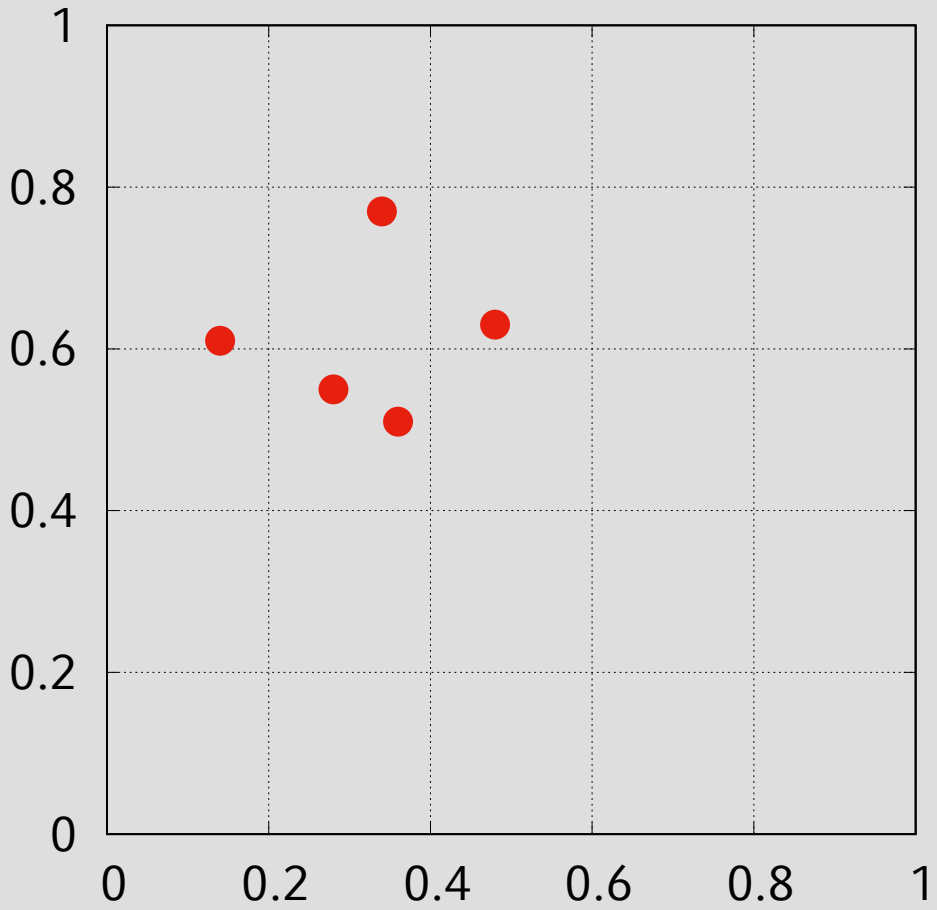
```
# -------------------------------
# no. particles in line = 10
particle position: ( 0.84 , 0.39 )
particle position: ( 0.78 , 0.80 )
particle position: ( 0.91 , 0.20 )
particle position: ( 0.34 , 0.77 )
particle position: ( 0.28 , 0.55 )
particle position: ( 0.48 , 0.63 )
particle position: ( 0.36 , 0.51 )
particle position: ( 0.95 , 0.92 )
particle position: ( 0.64 , 0.72 )
particle position: ( 0.14 , 0.61 )
# -------------------------------
```



Example: Particle Tracing (8/8)

now remove all points in right half:

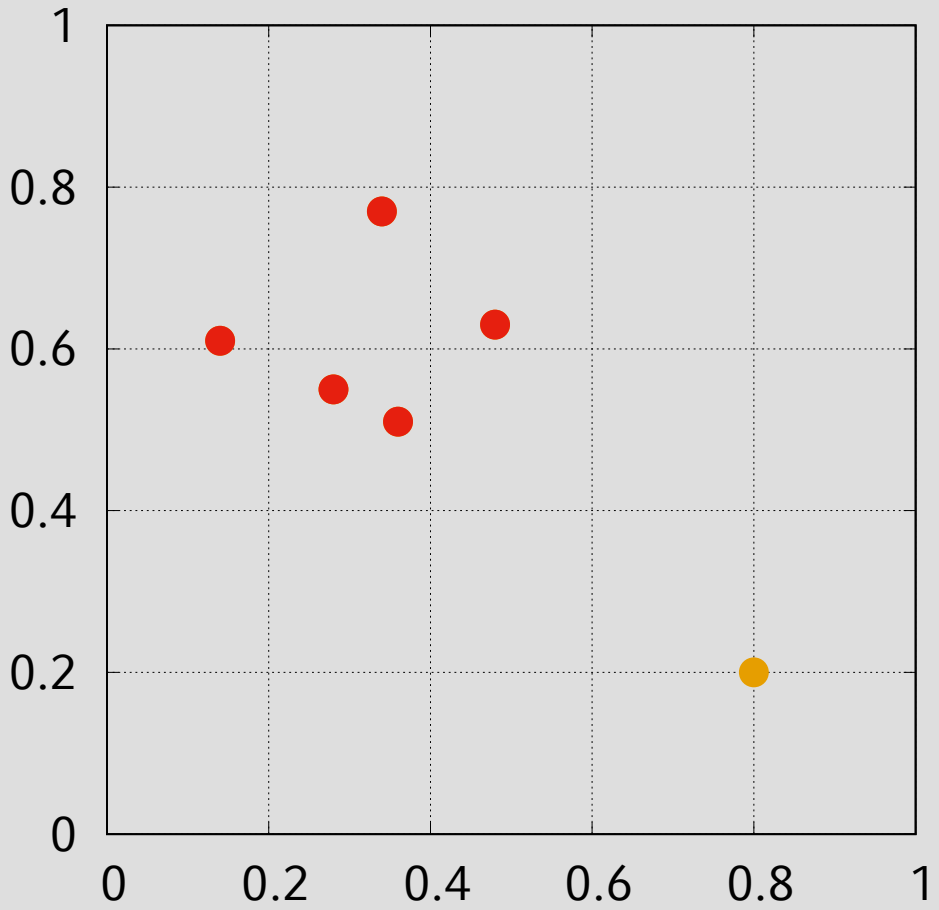
```
# ------------------------------
# no. particles in line = 5
particle position: ( 0.34 , 0.77 )
particle position: ( 0.28 , 0.55 )
particle position: ( 0.48 , 0.63 )
particle position: ( 0.36 , 0.51 )
particle position: ( 0.14 , 0.61 )
# ------------------------------
```



Example: Particle Tracing (8/8)

finally add a particle at $(0.8, 0.2)$ as last but one list entry:

```
# --------------------------------
# no. particles in line = 6
particle position: ( 0.34 , 0.77 )
particle position: ( 0.28 , 0.55 )
particle position: ( 0.48 , 0.63 )
particle position: ( 0.36 , 0.51 )
particle position: ( 0.80 , 0.20 )
particle position: ( 0.14 , 0.61 )
# --------------------------------
```



Sequence Containers (1/2)

Container

A container is a class, a data structure, or an abstract data type (ADT) whose instances are collections of other objects. (Wikipedia)

Sequence

A sequence $s = \langle e_0, e_1, \ldots, e_n \rangle$ is a collection of elements in a specific ordering. Same elements are allowed to appear multiple times at different positions in the sequence.

Sequence Containers (2/2)

We already know two types of sequence containers:

- singly linked list
  - allows to store a collection of elements in a specific ordering
  - does not require to know (maximal) length of sequence a priori
- array
  - access to element e_i of sequence s via `s[i]`
  - length known a priori: static array
  - maximal length known a priori: bounded array

Two special types of sequence containers are the stack and the queue.

Stacks (1/7)

- example: office clerk
  - has a stack of documents to process
  - new documents are put onto top of stack
  - next document to be processed also taken from top of stack
- LIFO principle (last in first out)
- two standard operations:
  - `push()` = put new element onto top of stack
  - `pop()` = remove top element from stack

Stacks (2/7)

- We implement a stack for values of type `elem_t` (`float` in demo)
- We use an array to implement a bounded stack (i.e. stack with a maximal capacity)
- The stack data structure will contain auxilliary management info:

```
typedef struct {
  elem_t* elem;
  int capacity;  // capacity of stack
  int top;       // index of top element
} stack_t;
```

- note that we will store the elements on the stack (involves copying), alternatively we could store their addresses (`elem_t**`)

Stacks (3/7)

- `initStack` will perform memory allocation
- and set the auxilliary info to sensible values
- we define macro `EMPTY_STACK` to be `-1`

```
void initStack( stack_t* stack, int capacity ) {
  stack->capacity = capacity;
  stack->top = EMPTY_STACK;
  stack->elem = malloc( capacity * sizeof( elem_t ) );
}
```

Stacks (3/7)

- `initStack` will perform memory allocation
- and set the auxilliary info to sensible values
- we define macro `EMPTY_STACK` to be `-1`

  (as this is no valid value for the top index of the stack)

```
void initStack( stack_t* stack, int capacity ) {
  stack->capacity = capacity;
  stack->top = EMPTY_STACK;
  stack->elem = malloc( capacity * sizeof( elem_t ) );
}
```

Stacks (4/7)

- we define two convenience macros to check the state of our stack
- using parameterised macros avoids overhead of external function call

```
#define IS_EMPTY( stack ) (stack->top == EMPTY_STACK)
#define IS_FULL( stack ) (stack->top == stack->capacity - 1)
```

- stack empty, if top element index is `EMPTY_STACK` (our convention: -1)
- stack full, if top element index equals (capacity-1)

- in object–oriented terminology `IS_EMPTY()` and `IS_FULL()` are called getter methods or accessor functions
- they return info on the status of the data-structure (object) but do not alter it.

Stacks (5/7)

- push places new element (item) onto top of stack
- can only push element on stack, if stack is not full!
- need to step top index as part of inserting item

```
void push( stack_t* stack, elem_t item ) {

  if ( IS_FULL( stack ) ) {
    fprintf( stderr, "stack capacity exceeded!\n" );
    exit( EXIT_FAILURE );
  }

  stack->top += 1;
  stack->elem[ stack->top ] = item;
}
```

Stacks (6/7)

- pop removes element from top of stack
- value is returned to caller
- can only pop element from stack, if stack is not empty!
- need to decrease top index (no explicit deletion)

```
elem_t pop( stack_t* stack ) {
  if ( IS_EMPTY( stack ) ) {
    fprintf( stderr, "cannot pop from empty stack!\n" );
    exit( EXIT_FAILURE );
  }
  stack->top -= 1;
  return stack->elem[ stack->top + 1 ];
}
```

Stacks (7/7)

```
stack_t stack = { NULL, 0, 0 };
elem_t aux;

initStack( &stack, 3 );

// operations on stack
push( &stack, 1.0 );
push( &stack, 2.0 );
push( &stack, 3.0 );

aux = pop( &stack );

push( &stack, 4.0 );

aux = pop( &stack );
aux = pop( &stack );
aux = pop( &stack );

clearStack( &stack );
```

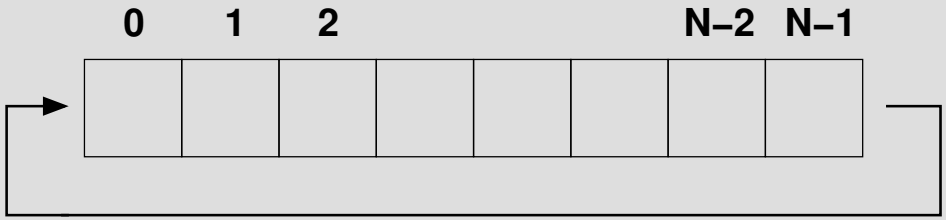
top index	contents of stack
-1	(empty) (before first push)
0	1.0
1	1.0 2.0
2	1.0 2.0 3.0
1	1.0 2.0
2	1.0 2.0 4.0
1	1.0 2.0
0	1.0
-1	(empty) (after last pop)

Queues

- example: post office
  - customers start queueing at end of line (rear/tail)
  - first customer in line (front/head) get's served
- FIFO principle (first in first out)
- two standard operations:
  - `enqueue()` = append new element at tail
  - `dequeue()` = remove element from head
- bounded FIFO queue can be implemented with circular array

Circular Array

- A circular array is just a different interpretation of a standard array.
- We imagine the array being closed, i.e. after the last entry we get to the first again



- Array indices outside $\mathcal{I} = \{0, 1, \ldots, N-1\}$ can be mapped onto $\mathcal{I}$ by a modulo operation (k mod N).

Queue Example

operation	length	head	tail	queue elements	
`enqueue( 1.0 )`	1	0	0	`1.0 *** *** ***`	initial queue empty
`enqueue( 2.0 )`	2	0	1	`1.0 2.0 *** ***`	
`enqueue( 3.0 )`	3	0	2	`1.0 2.0 3.0 ***`	
`enqueue( 4.0 )`	4	0	3	`1.0 2.0 3.0 4.0`	
`dequeue()`	3	1	3	`*** 2.0 3.0 4.0`	
`dequeue()`	2	2	3	`*** *** 3.0 4.0`	
`dequeue()`	1	3	3	`*** *** *** 4.0`	
`enqueue( 5.0 )`	2	3	0	`5.0 *** *** 4.0`	wrap–around
`enqueue( 6.0 )`	3	3	1	`5.0 6.0 *** 4.0`	
`dequeue()`	2	0	1	`5.0 6.0 *** ***`	
`dequeue()`	1	1	1	`*** 6.0 *** ***`	
`enqueue( 7.0 )`	2	1	2	`*** 6.0 7.0 ***`	
`dequeue()`	1	2	2	`*** *** 7.0 ***`	

Queue Implementation (1/2)

- We store queue elements in an array `queue(0:capacity-1)`. Using 0-based indexing for ease of modulo operation.
- We employ this queue data structure with management info

```
typedef struct {
  elem_t* elem;
  int capacity;    // capacity of queue
  int head;        // index of head of queue
  int tail;        // index of tail of queue
  int length;      // current length of queue
} queue_t;
```

- Current queue elements reside in `elem[head]` $\leftrightarrow$ `elem[tail]`, including possibly a wrap–around.

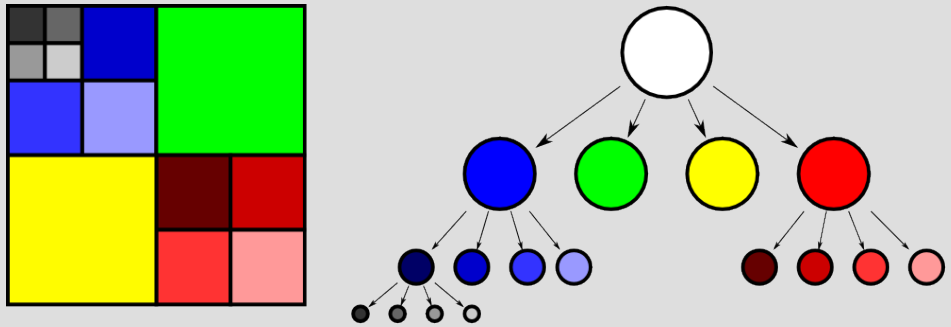
Queue Implementation (2/2)

- as an example we consider `enqueue()` routine.
- `item` contains new value to be added after tail
- for this we increment tail and perform a modulo operation on it:

$$k \bmod q_{\text{cap}} \in \{0, 1, \ldots, q_{\text{cap}} - 1\}$$

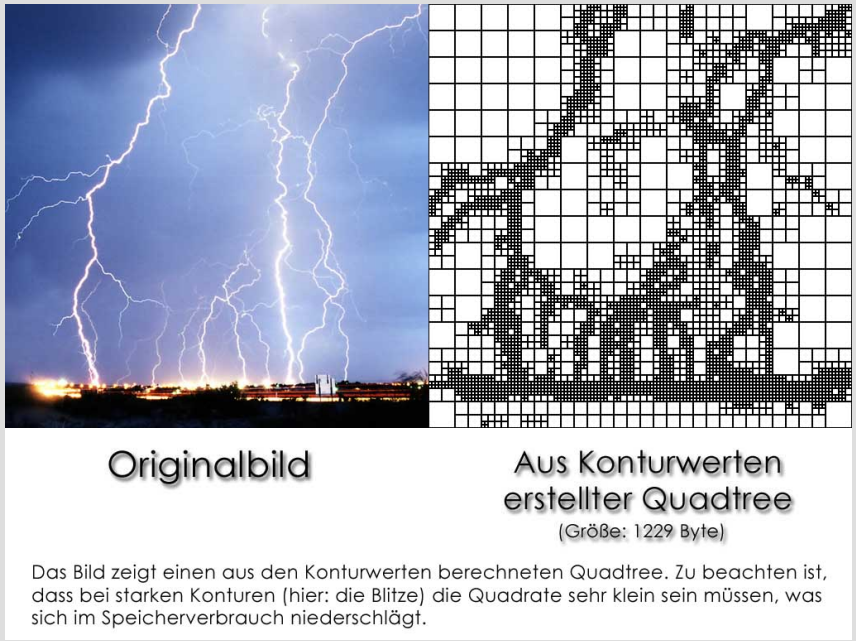
```
void enqueue( queue_t* queue, elem_t new ) {
  if( queue->length < queue->capacity ) {
    queue->tail = (queue->tail + 1) % queue->capacity;
    queue->elem[ queue->tail ] = new;
    queue->length += 1;
  }
  else {
    fprintf( stderr, "queue capacity exceeded!\n" );
    exit( EXIT_FAILURE );
  }
}
```

QuadTrees

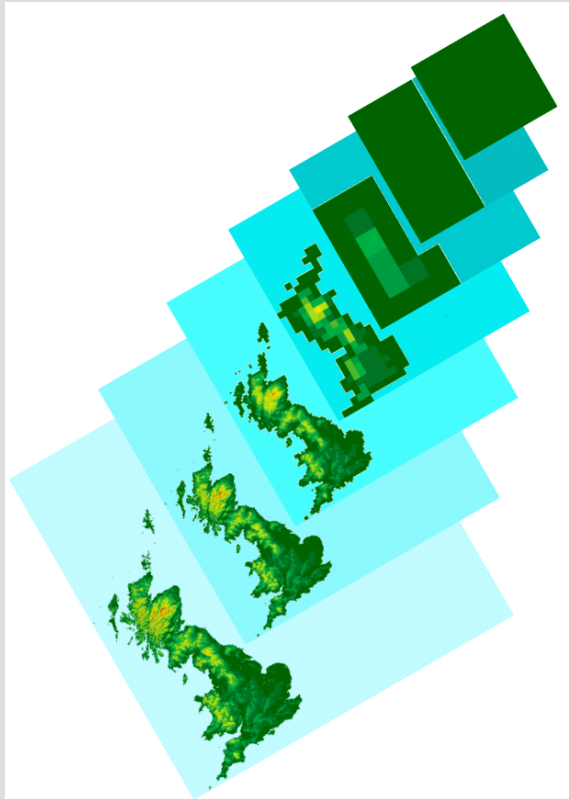


- a quadtree is a specialised form of tree structure
- each node of the tree is either a leaf (no children) or has four children
- identical concept for 3D is an octree
- used e.g. in image processing, area indexing (GIS), collision detection, hidden surface removal of terrain data, mesh generation

QuadTrees — Examples



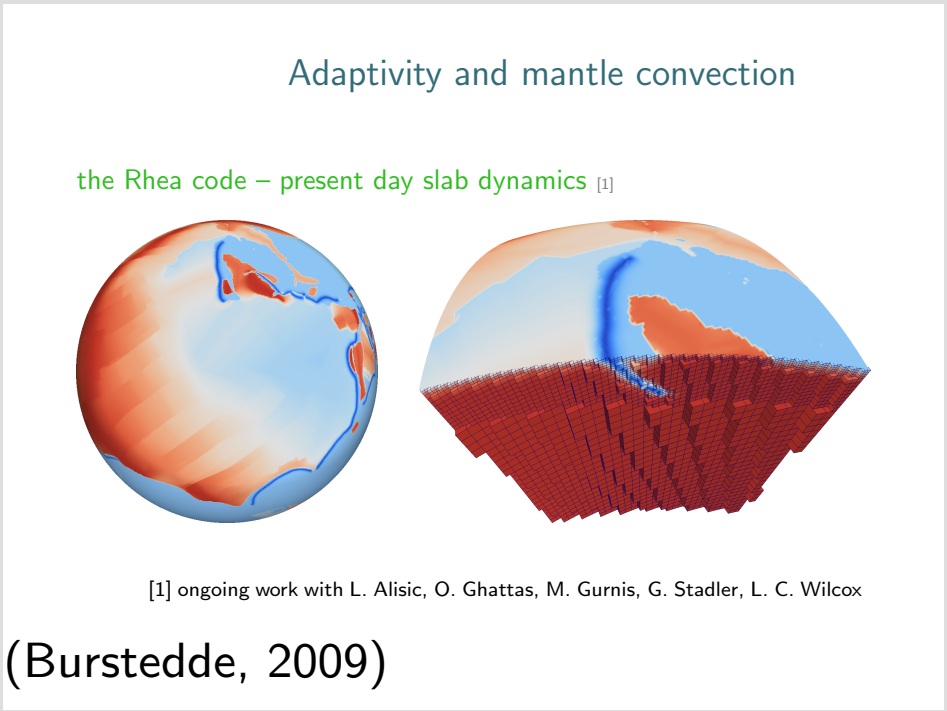
Lightning



Great Britain

Kidner et al.

Octrees — Example

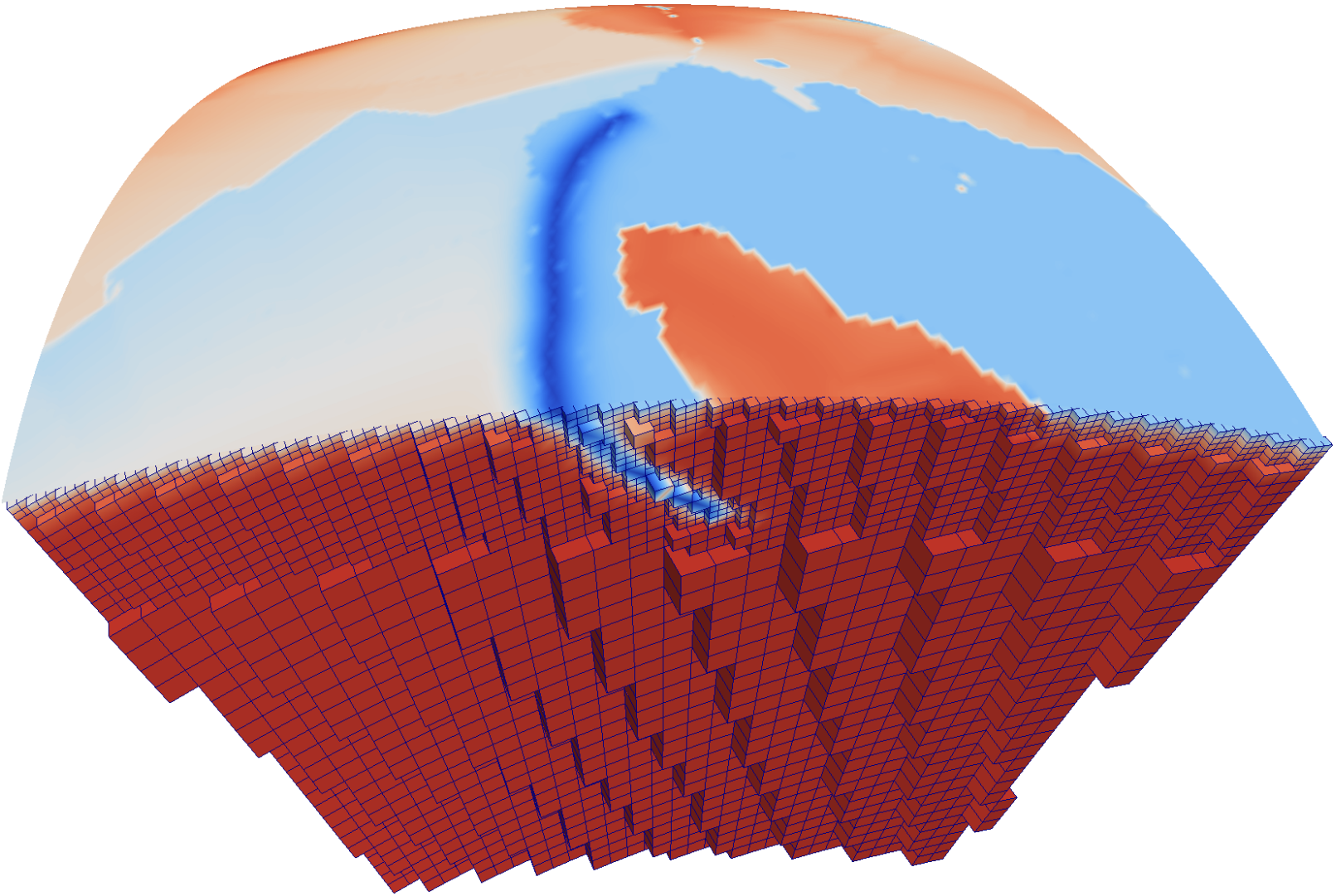


(Burstedde, 2009)

- Octrees are the 3D extension of quadtrees
- are used in many applications such as
  - spatial indexing
  - efficient collision detection in three dimensions
  - view frustum culling
  - fast Multipole Method
  - unstructured grid
  - finite element analysis

Octrees — Example

slab dynamics [1]



Dense versus Sparse Matrices

(Weak) Definition:

A matrix is called sparse when the number of non-zero matrix entries is significantly smaller than the number of zero matrix entries.

On the algorithmic side sparse matrices pose completely different challenges than dense matrices.

Dense versus Sparse Matrices

(Weak) Definition:

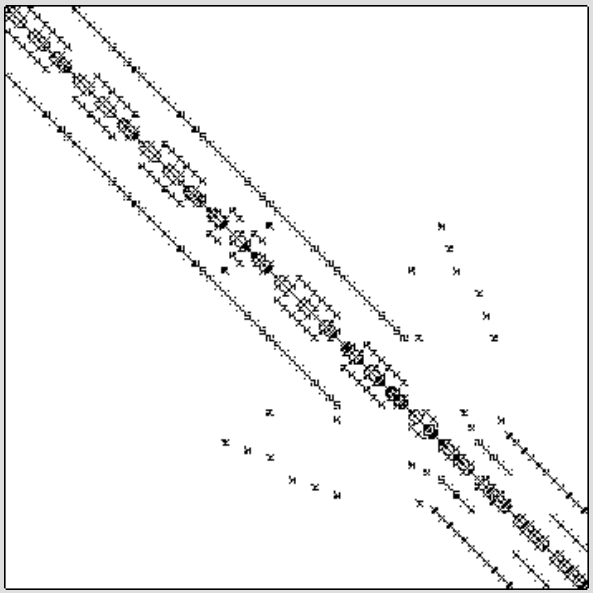
A matrix is called sparse when the number of non-zero matrix entries is significantly smaller than the number of zero matrix entries.

On the algorithmic side sparse matrices pose completely different challenges than dense matrices. E.g.

- Storing a sparse matrix as 2D-array is not economic (due to the large number of zeros) and in most cases not even feasible
- Fast methods for solving $Ax = b$ must take the sparsity into account, i.e. you only want to deal with the non-zero entries

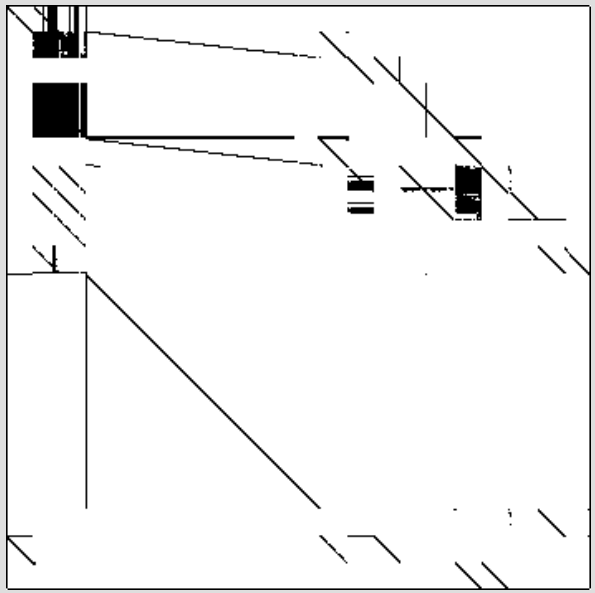
Examples from the MatrixMarket

`http://math.nist.gov/MatrixMarket/`



Structural Engineering

(NOS5)



Economic Modelling

(ORANI678)

Weather Simulation

(FS 760 3)

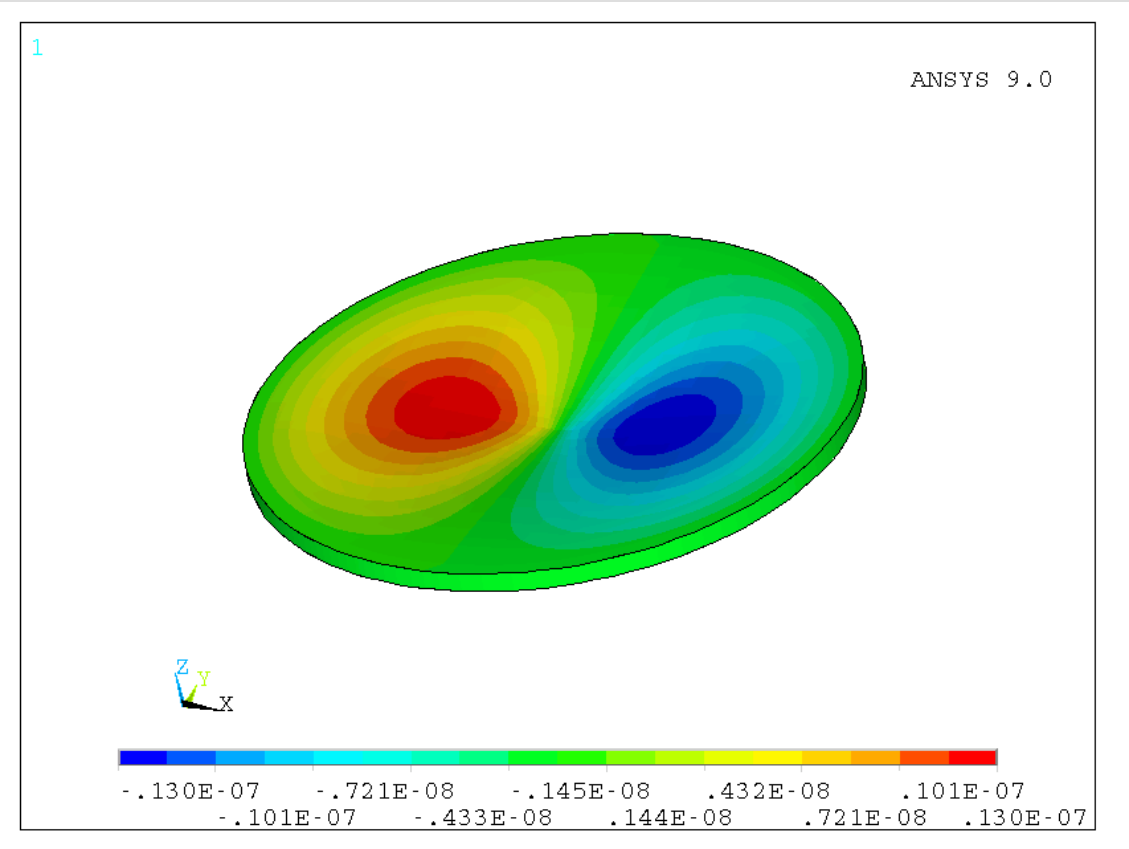
Where do Sparse Matrices come from?

A major source of sparse matrices is the discretisation of partial differential equations.

In practice the four most important discretisation methods are:

Finite Differences	sparse matrix	volume discretisation techniques
Finite Elements	sparse matrix	
Finite Volumes	sparse matrix	
Boundary Elements	dense matrix	surface discretisation technique

Example: Structural Mechanics



$u_x = u_y = u_z = 0$
on boundary

load in centre

standard discretisation with tri-linear Finite elements on hexaeders

# nodes	3,726
# elements	2,370
n	10,170
nnz	619,038
sparsity	0.60%

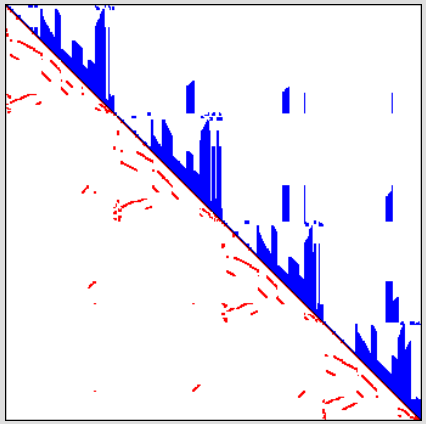
Example: Re-ordering

Cholesky decomposition $A = LL^T$ with $\dim(A) = 10,170$

non-zeros to store $A = 314,604$

Example: Re-ordering

Cholesky decomposition $A = LL^T$ with $\dim(A) = 10,170$
non-zeros to store $A = 314,604$

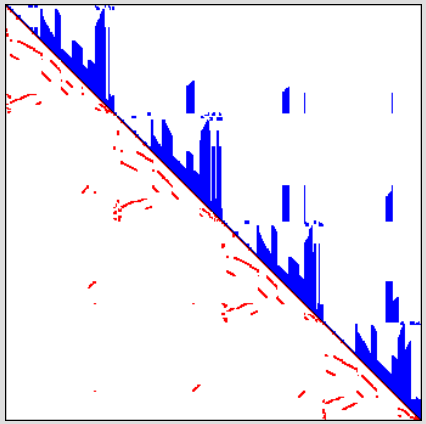


'original'

non-zeros in L
5,749,983
increase by
18.3

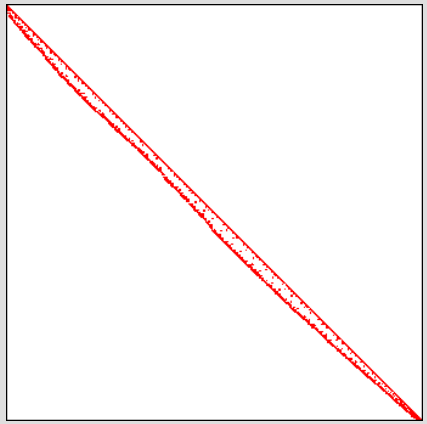
Example: Re-ordering

Cholesky decomposition $A = LL^T$ with $\dim(A) = 10,170$
non-zeros to store $A = 314,604$



'original'

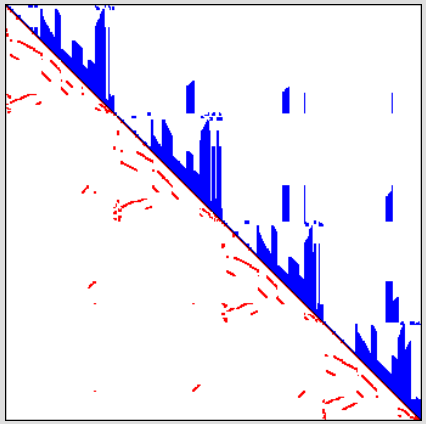
non-zeros in L
5,749,983
increase by
18.3



Cuthill-McKee

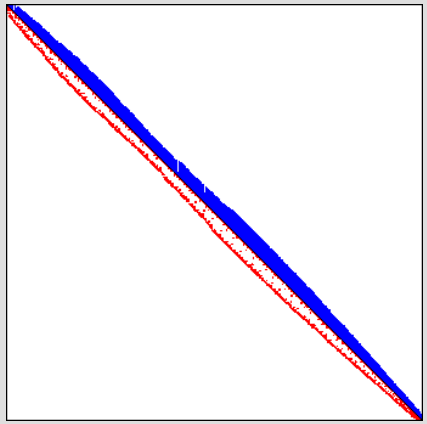
Example: Re-ordering

Cholesky decomposition $A = LL^T$ with $\dim(A) = 10,170$
non-zeros to store $A = 314,604$



'original'

non-zeros in L
5,749,983
increase by
18.3

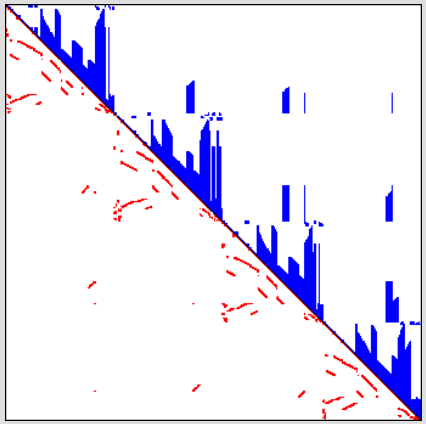


Cuthill-McKee

non-zeros in L
3,581,856
increase by
11.4

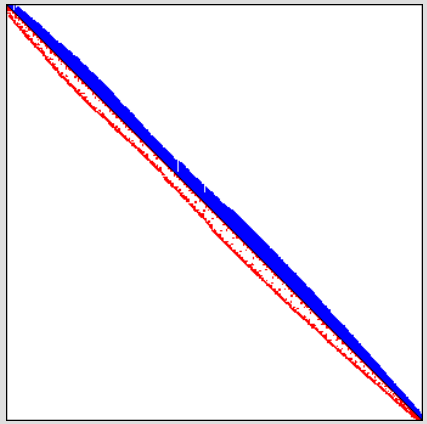
Example: Re-ordering

Cholesky decomposition $A = LL^T$ with $\dim(A) = 10,170$
non-zeros to store $A = 314,604$



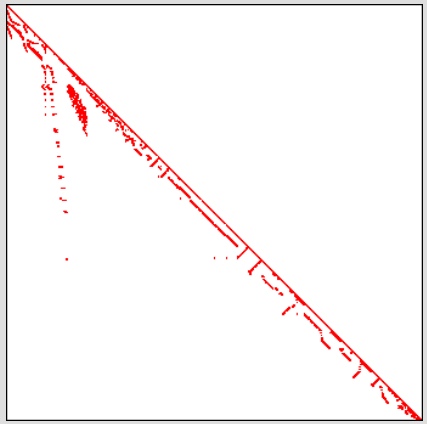
'original'

non-zeros in L
5,749,983
increase by
18.3



Cuthill-McKee

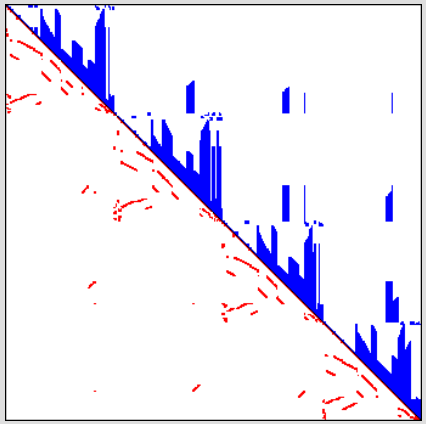
non-zeros in L
3,581,856
increase by
11.4



Sloan

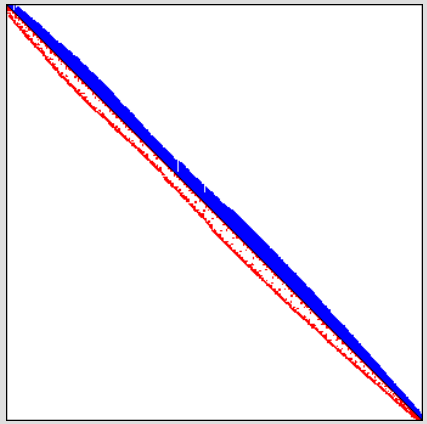
Example: Re-ordering

Cholesky decomposition $A = LL^T$ with $\dim(A) = 10,170$
non-zeros to store $A = 314,604$



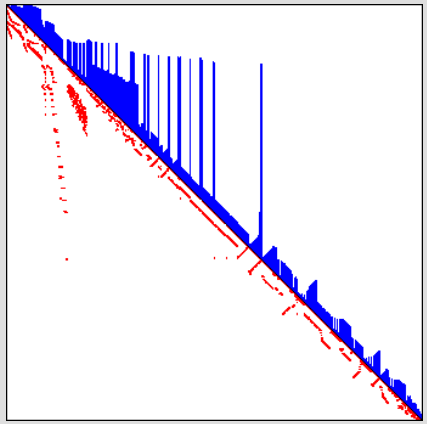
'original'

non-zeros in L
5,749,983
increase by
18.3



Cuthill-McKee

non-zeros in L
3,581,856
increase by
11.4

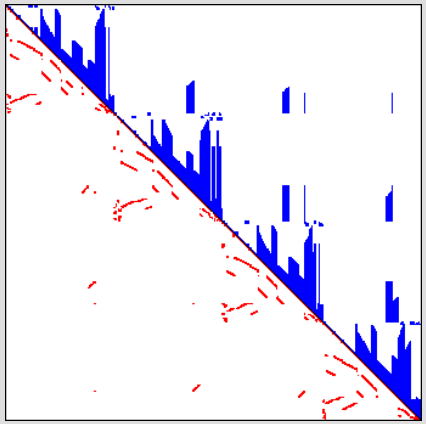


Sloan

non-zeros in L
4,133,448
increase by
13.1

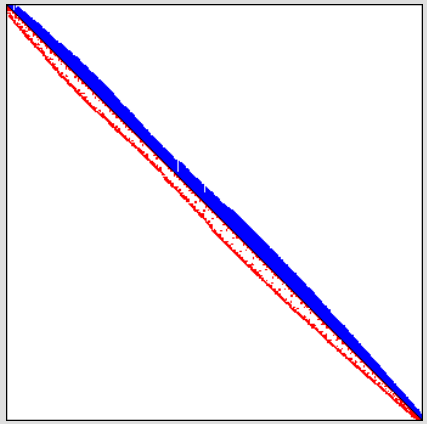
Example: Re-ordering

Cholesky decomposition $A = LL^T$ with $\dim(A) = 10,170$
non-zeros to store $A = 314,604$



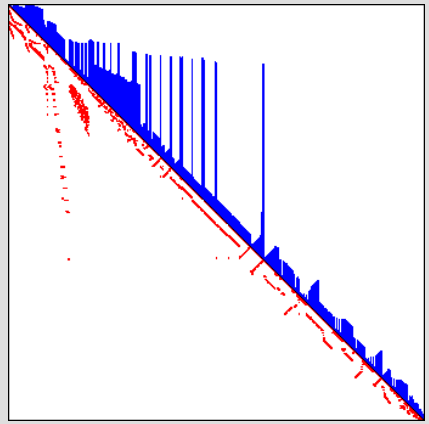
'original'

non-zeros in L
5,749,983
increase by
18.3



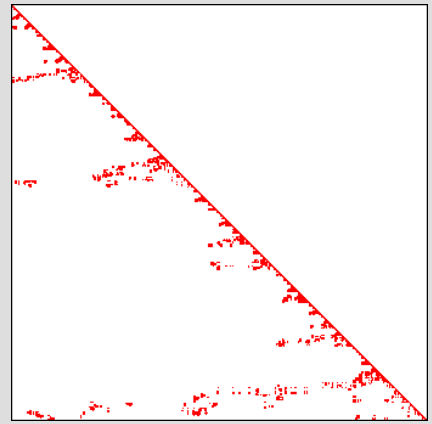
Cuthill-McKee

non-zeros in L
3,581,856
increase by
11.4



Sloan

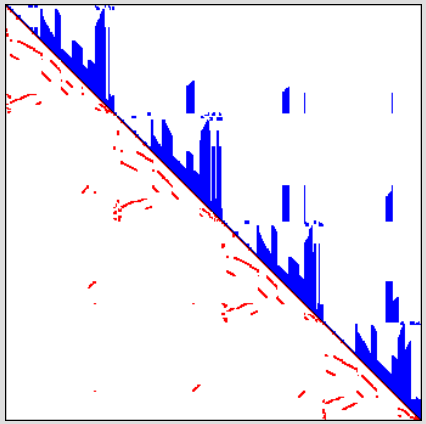
non-zeros in L
4,133,448
increase by
13.1



minimum degree

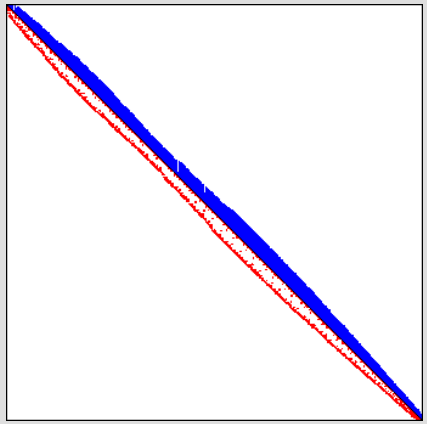
Example: Re-ordering

Cholesky decomposition $A = LL^T$ with $\dim(A) = 10,170$
non-zeros to store $A = 314,604$



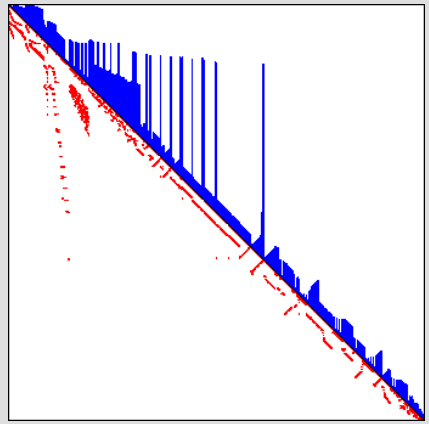
'original'

non-zeros in L
5,749,983
increase by
18.3



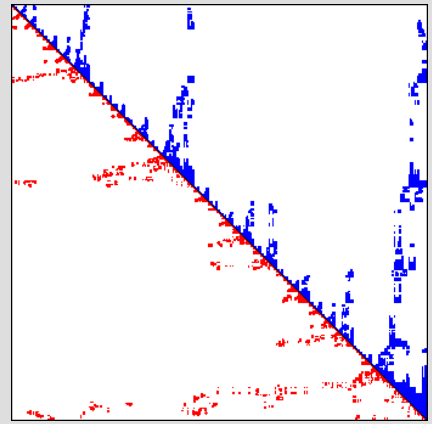
Cuthill-McKee

non-zeros in L
3,581,856
increase by
11.4



Sloan

non-zeros in L
4,133,448
increase by
13.1



minimum degree

non-zeros in L
2,029,509
increase by
6.5

Practical Issues

The question how to store a sparse matrix turns out to be non-trivial.

- storing it as a 2D array $\longleftarrow$ impossible
- answer strongly depends on what you want to do with it

Approach I: Index scheme

For each non-zero entry a_{ij} store the triplet:

row index i	column index j	non-zero value a_{ij}

- we need three arrays of length nnz (number of non-zeros)
- each triplet requires (4+4+8) byte per entry (on typical 32-bit architecture)
- total storage costs: $16 \cdot \text{nnz}$ bytes

Index scheme (cont.)

Matrix from Structural Mechanics example

```
%%MatrixMarket matrix coordinate real symmetric
%
% Matrix exported by OLAS
%
10170   10170   314604
     1       1   8.2990067682228677e+04
     1       2  -4.9696384386100754e+03
     1       3  -2.7606714995454240e-01
     1       4   9.7198803223981122e+03
     1       5  -6.1171895993969656e+03
     1       6  -2.7255939150670642e+03
     1       7  -1.7210872534276655e+03
     1       8   2.5811602286800858e+03
     1       9   8.7842380729450872e+02
```

requires 9.4 MB of storage

problem with index scheme is that it does not give any structural support, e.g. no indication where a row starts

(think of matrix-vector product $A \cdot u$)

we can also do a little better on storage

Approach II: Compressed Row Storage

$$\begin{bmatrix} -4 & 8 & 7 & . \\ 5 & -6 & . & 2 \\ 3 & . & -1 & . \\ . & 10 & . & -9 \end{bmatrix}$$

CRS format employs three arrays:

		1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
val	=	-4	8	7	5	-6	2	3	-1	10	-9
cidx	=	1	2	3	1	2	4	1	3	2	4
rptr	=	1	4	7	9	11					

Compressed Row Storage (cont.)

The idea of the compressed row storage format is to:

- to store for all non-zero entries their value and column index
- entries for a row are stored contiguously (but in no special order)
- rows are stored one after the other lexicographically
- for each row we have a 'pointer' where the row starts in the other two arrays

Compressed Row Storage (cont.)

- CRS requires a little less memory:

array	contents	type	length
val	values	(double)	nnz
cidx	column indices	(integer)	nnz
rptr	row start indices	(integer)	n + 1

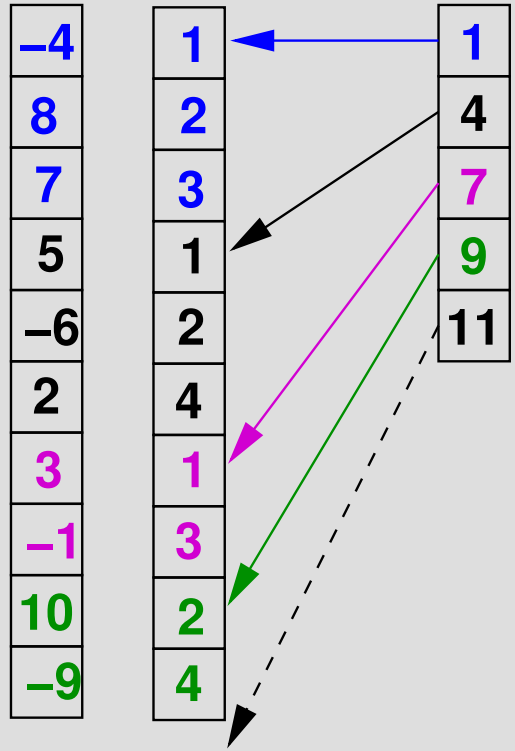
$\longrightarrow (8+4)\text{nnz} + 4(n+1)$ ($\longrightarrow$ 7.1 MB in example)

- CRS provides structural information
  - we know where a row starts
  - we know how many non–zeros are in row k from rptr[k+1] - rptr[k]

Matrix-Vector Product with CRS

The product $Au = v$ can be implemented as

```
for i = 1 to n do
    sum = 0
    k = rowPtr[i]
    rs = rowPtr[i+1] - k
    for j = 1 to rs do
        sum = sum + val[k] · u[cidx[k]]
        k = k + 1
    end for
    v[i] = sum
end for
```



Other Sparse Storage Formats

There is a multitude of different storage formats for sparse matrices

- CRS sub-variants (like storing the diagonal entry in first position for each row)
- Compressed Column Storage (CCS)
- Compressed Diagonal Storage (CDS)
- Jagged Diagonal Storage (JDS)
- Skyline Storage

which all have their pros and cons.

Resources

- Bebis, Univ. of Nevada, CS308: Data Structures
- Morris, Univ. of Auckland, PLDS210: Programming Languages and Data Structures
- Wikipedia entry on *Data Structure* and associated entries
- Barrett, Berry, Chan et al., 'Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods', SIAM, 1994 (www.netlib.org)