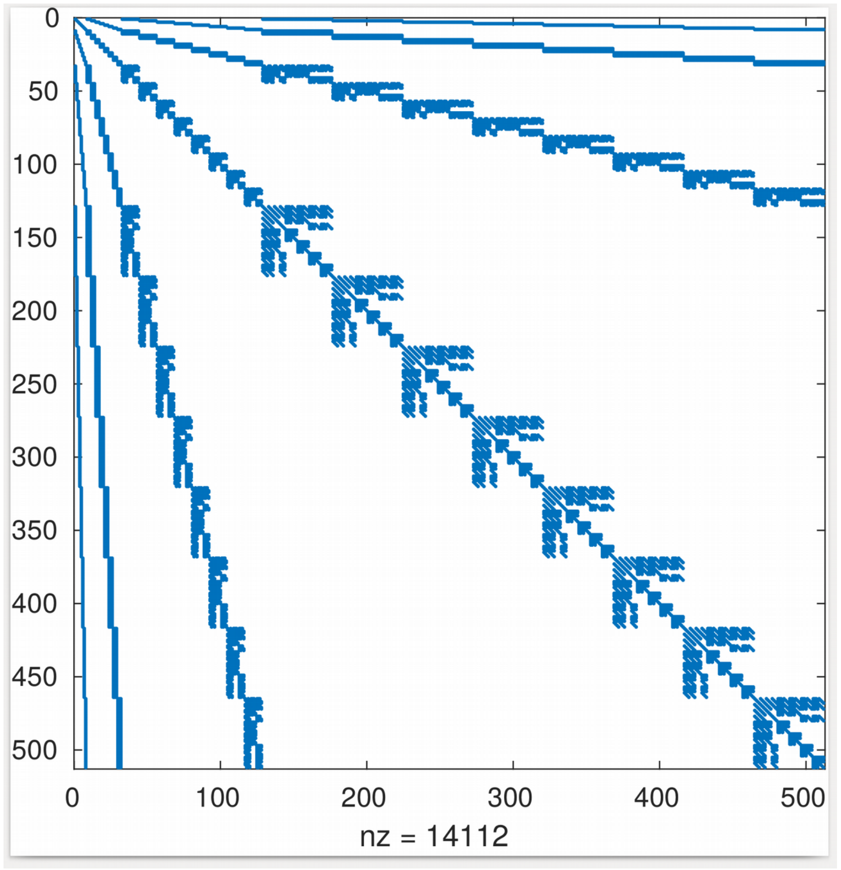
Lecture 8: Back to inheritance!

- Last lecture we extended our Matrix class to use templating
- That allows us to build our Matrix class using doubles, ints or other numeric types that have a * and [] operators defined
- We’re going to further extend our Matrix “library” through inheritance

Sparse matrices

- Our existing Matrix class stored dense matrices
- We spoke a little about different types of “sparse” matrices
- These are matrices where some entries are zero



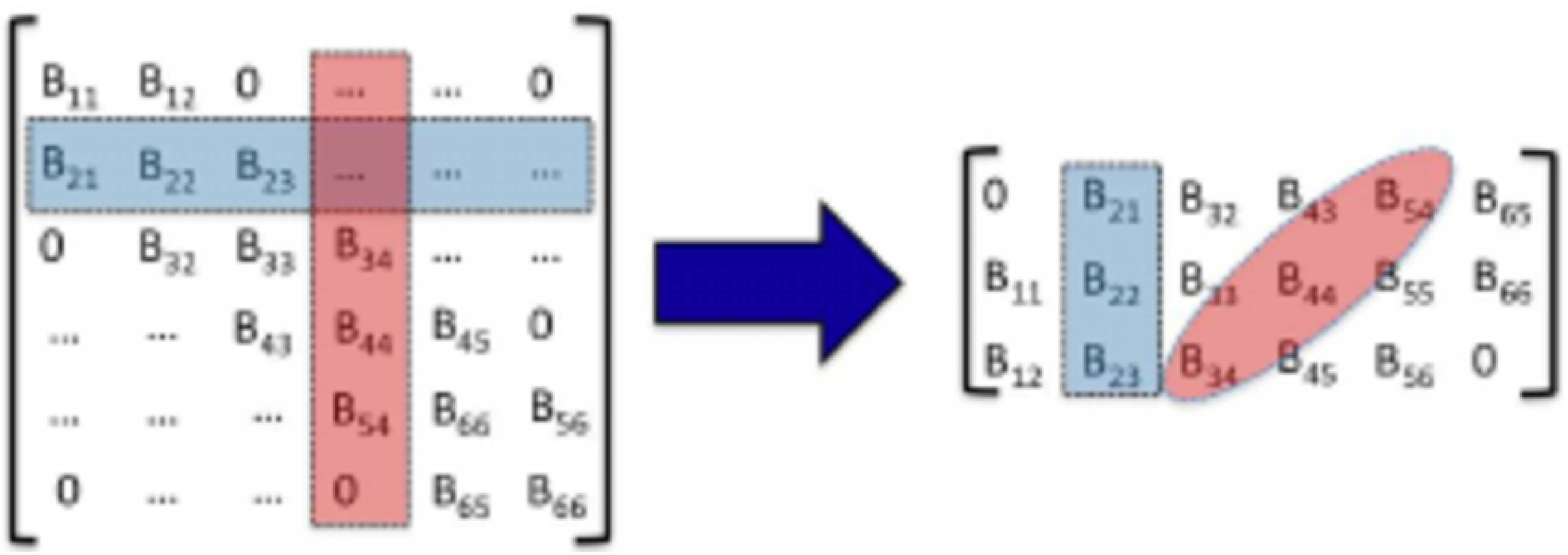
Banded matrices

- For when non-zero entries are a set amount away from the diagonal
- Very common way to store matrices that come from (local) spatial discretisations of PDEs (e.g., see ACSE-3 Lecture 7)

$$\begin{pmatrix} X & X & X & \cdot & \cdot & \cdot & \cdot \\ X & X & \cdot & X & X & \cdot & \cdot \\ X & \cdot & X & \cdot & X & \cdot & \cdot \\ \cdot & X & \cdot & X & \cdot & X & \cdot \\ \cdot & X & X & \cdot & X & X & X \\ \cdot & \cdot & \cdot & X & X & X & \cdot \\ \cdot & \cdot & \cdot & \cdot & X & \cdot & X \end{pmatrix}$$

Banded matrices

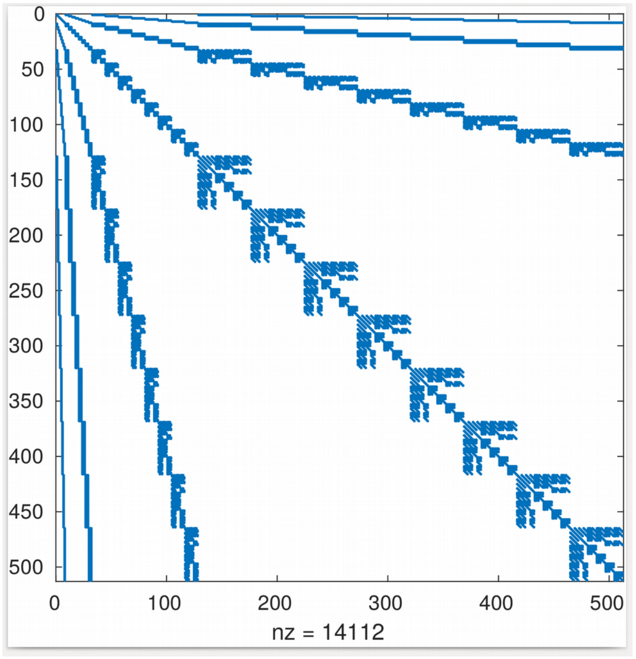
- How would we store a banded matrix?
- How would we do a matrix-matrix product in that format?

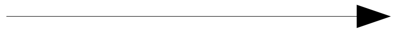


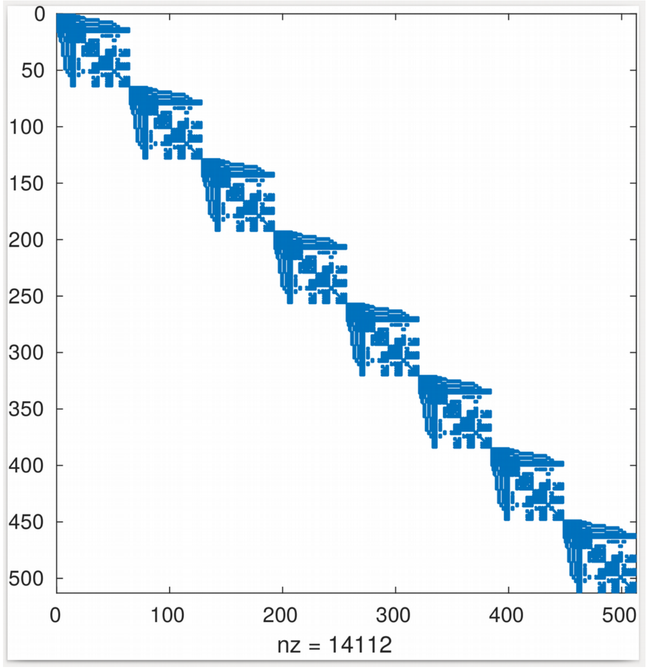
B_{11} B_{12} 0 0
B_{21} B_{22} B_{23} ... ...
0 B_{32} B_{33} B_{34} ... ...
... ... B_{43} B_{44} B_{45} 0
... ... B_{54} B_{66} B_{56}
0 ... ... 0 B_{65} B_{66}
0 B_{21} B_{32} B_{43} B_{54} B_{65}
B_{11} B_{22} B_{33} B_{44} B_{55} B_{66}
B_{12} B_{23} B_{34} B_{45} B_{56} 0

Algorithms for changing sparsity

- The sparsity of a matrix is a consequence of the ordering of the unknowns
- The solution is still the same
- Algorithms exist to reorder the unknowns in desirable ways
- E.g., RCM (reverse Cuthill-McGee) to reduce bandwidth







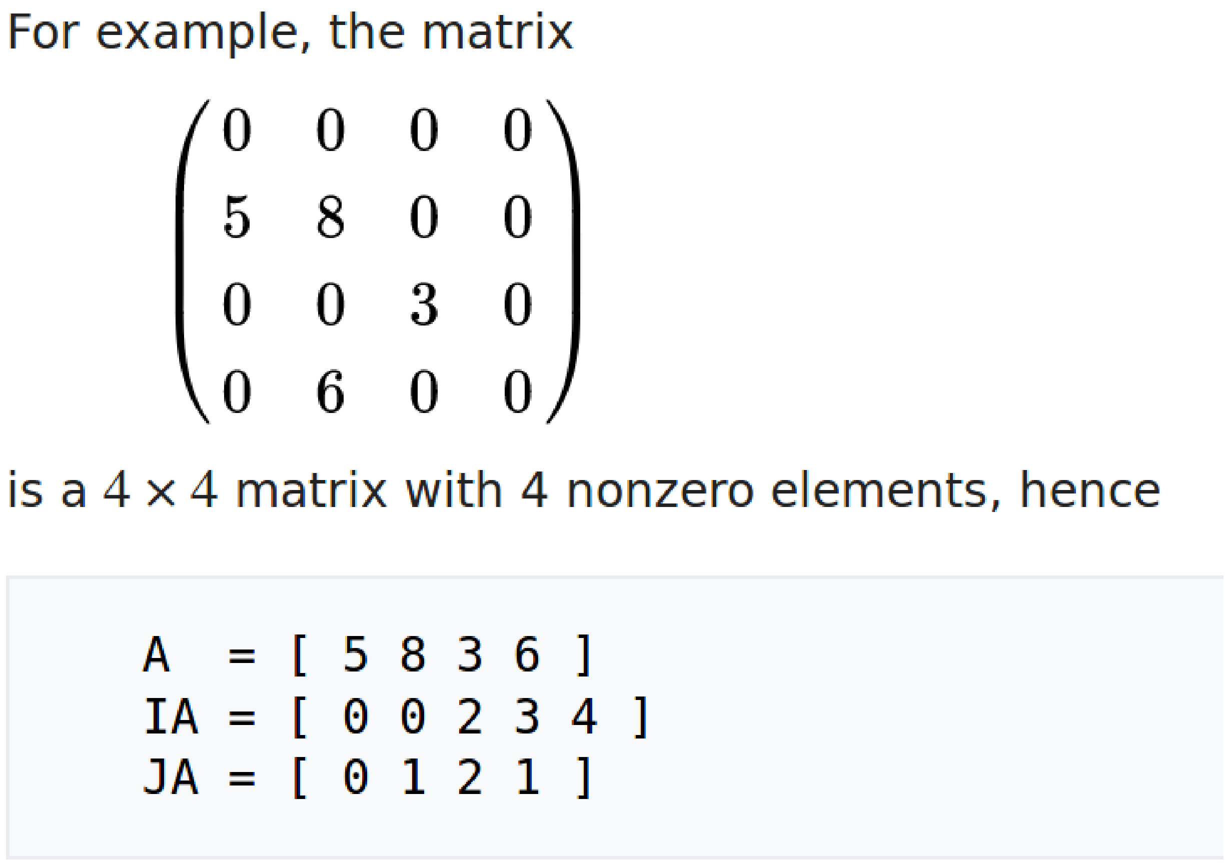
Symmetric matrices

- Symmetry can be exploited to reduce the storage, either in dense, banded or CSR storage of a symmetric matrix
- Not quite half the storage (given the diagonal)

$$A = \begin{pmatrix} 1 & -1 & * & -3 & * \\ -1 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -3 & * & 6 & 7 & * \\ * & * & 4 & * & -5 \end{pmatrix}$$

CSR matrices

- Known as compressed sparse row
- A store the values, IA the index into each row, JA the column for each non-zero



For example, the matrix

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 5 & 8 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 6 & 0 & 0 \end{pmatrix}$$

is a 4×4 matrix with 4 nonzero elements, hence

```
A  = [ 5 8 3 6 ]
IA = [ 0 0 2 3 4 ]
JA = [ 0 1 2 1 ]
```

Time to write some code!

- Get your Matrix.h and Matrix.cpp from last lecture
- Now it’s time to examine some design decisions we made when we created Matrix.h and Matrix.cpp
- Why was “preallocated” private?
- Why did we make matmatMult virtual?
- We are going to create CSRMatrix.h and CSRMatrix.cpp

Matrix-vector product

- How easily vectorised do you think the matrix-vector product we wrote is?

```
output[i] += this->values[val_index] *
    input[this->col_index[val_index]];
```

CSRMatrix

- Let’s think about our matmatMult(mat_right, output)
- How do we know the sparsity of our output CSRMatrix
- Often sparse libraries will compute a “symbolic matmatMult”
- This just loops over the sparsity as if it were doing a matrix matrix product
- A = B * C
- If there aren’t entries in the corresponding sparsity of B and C, there won’t be a non-zero entry in A
- If the sparsity doesn’t change between matmatMults, you don’t have to redo the symbolic

Existing libraries

- There are lots of sparse matrix libraries (that also include lots that already exist
- Rule number 1: Never write code if someone clever has already done the hard work for you!
- PETSc (Portable Extensible Toolkit for Scientific Computation) (https://www.mcs.anl.gov/petsc/)
- Trilinos (github.com/trilnos/Trilinos)
- Intel MKL

PETSc

- Heavily used and supported
- Lots of different dense/sparse Matrix classes
- Lots of different linear solvers (including direct/iterative solvers)
- For example look at all the different Matrix methods they provide:
- The serial PETSc CSR class is called MATAIJ

https://www.mcs.anl.gov/petsc/petsc-current/docs/manualpages/Mat/index.html

PETSc methods to consider:

- MatCreateAIJ
- MatSetValue
- MatBeginAssembly
- MatEndAssembly
- MatMatMultSymbolic
- MatMatMultNumeric
- We can take inspiration from these methods
- “Under the hood” a lot of these methods make calls to BLAS/LAPACK!

Homework

- Rewrite our CSRMatrix::printMatrix to print out in a 2D pattern
- Build a setvalues() function for our CSRMatrix inspired by PETSc that takes in i,j, and a value, and sets the value in that position (maybe include something like “Assemble”)
- ADVANCED - Build a symbolic matrix-matrix product for our CSRMatrix
- SUPER ADVANCED – Build our matMatMult for CSRMatrix, which calls the symbolic matrix-matrix product to know how many nnzs to allocate for the output
- ULTRA ADVANCED – Try using PETSc, either install/link and call from C++, or use PETSc4Py on a Jupyter notebook