ECE 421
Assignment 1 Design Report

By: Vishal Patel, Jose Ramirez, Rizwan Qureshi, Curtis Goud, Jori Romans

Design Rationale:

We designed our sparse matrix library with the goal of providing computational optimization for sparse matrices when performing matrix operations. We chose to support:

- Basic scalar and vector matrix operations
- Operator overloading (eg. Assume A and B are a matrix of nxm elements, our library supports A+B, A-B, A*B, etc.)
- Custom iterators that allow you to parse specific columns or rows.
- Support for “list of lists”, “dictionary of keys” and the new “Yale” storage types
- Conversion between all matrix storage types
- LU decomposition and Eigenvalue decomposition
- Construction and checking of special matrix types: Zero matrix, Identity matrix, Tridiagonal matrix

This library has a “list of lists” and “dictionary of keys” implementation for storage of matrices where efficient modification is required and a new “Yale” implementation for storage of matrices where sufficient access and matrix operations are required. We plan to create an abstract factory to create matrix factories for each storage type. Our SMatrix library will contain the contracts and a general implementation of matrix operations and construction using the delegate design pattern which will delegate matrix operations to the respective implementation for that storage type.

To supplement our design, we chose to use the NMatrix gem that supports a large amount of features which include but not limited to: shorthand notation for matrix creation, list of lists and yale matrix storage, multiple data types, many element wise and matrix operations and custom iterators for Yale matrices. The main advantage to NMatrix is that it supports most of the features we chose to implement which will save us development time. The disadvantage to NMatrix is that a couple important matrix solving functions such as Eigenvalue decomposition is not available (therefore we may have to implement it ourselves). We also chose to implement the Dictionary of Keys method as a new standalone class, choosing to implement our own versions of the operations for this storage type.

System Research

1. What is a sparse matrix and what features should it possess?
    - A matrix is sparse if most of the elements are zero.
    - Sparsity of a matrix = (# of 0-valued elements) / (total # of elements) = 1 - (density of a matrix). A matrix is “sparse” if sparsity > 0.5

2. What sources of information should you consult to derive features? Do analogies exist? If so with what?
    - Analogy 1: Consider a line of balls connected by strings. If each adjacent ball was connected by a spring then the system is sparse. If each ball had a spring connecting it with all other balls than the system is dense.
    - Sources of Information:
        - Consulting Wikipedia (https://en.wikipedia.org/wiki/Sparse_matrix)
        - Textbook: Selected Chapters from Elementary Linear Algebra, Eleventh Edition by: Anton & Rorres
        - Existing linear algebra library: https://github.com/SciRuby/nmatrix . Has an experimental implementation in Ruby.
        - Existing implementations of Sparse Matrix libraries - https://www.mathworks.com/help/matlab/math/sparse-matrix-operations.html#f6-14457
        - Default matrix library for Ruby: https://docs.ruby-lang.org/en/2.4.0/Matrix.html

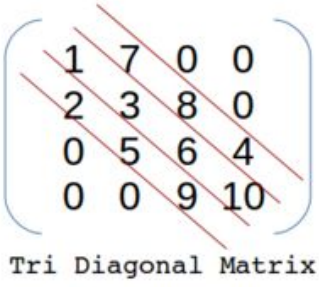
3. Who is likely to be the user of a sparse matrix package? What features are they likely to demand?
    - Sparse matrices correspond to loosely coupled systems and are used in machine learning, combinatorics and network theory which have a low density of significant data. Additionally, large sparse matrices appear in scientific and engineering applications when solving partial differential equations.
    - For machine learning, sparse matrices are usually used to store large quantities of training data. A user for this purpose would likely want efficient storage and iteration of a large sparse matrix.
    - For computer graphics, It is many operations of matrices multiplying themselves.
    - For Google or Amazon, everytime they recommend you something they multiply sparse matrices to yield that result for you.
    - For Google’s Adsense, they calculate positive eigenvectors to determine which advertisements to show you.
    - For information retrieval, the index is essentially an intelligently crafted sparse matrix.

- Other features that are common to other matrix libraries and therefore likely to be demanded are:
    - Basic matrix scalar and vector operations including: Add, subtract, multiply, transposition, trace, exponents, inverse, determinants,
    - Slice assignments, enumerators for matrices
    - Matrix and vector storage in multiple formats
    - Interconversion between storage and data types
    - Matrix slicing by copy and reference
    - Checking tridiagonal, identity, symmetric
    - Creation of common matrix types such as Identity, Zero, or Tridiagonal
    - LU and Eigenvalue factorization

4. What is a tri-diagonal matrix?
    - A tri-diagonal matrix is a matrix such that all non zero elements exist on the main diagonal, the diagonal existing one below the main diagonal, and one above the main diagonal.

$$f_n = \begin{vmatrix} a_1 & b_1 & & & \\ c_1 & a_2 & b_2 & & \\ & c_2 & \ddots & \ddots & \\ & & \ddots & \ddots & b_{n-1} \\ & & & c_{n-1} & a_n \end{vmatrix}.$$

5. What is the relationship between a tri-diagonal matrix and a generic sparse matrix?
    - Tri-diagonal matrices share the same property with sparse matrices which is that they are filled with many zero elements and relatively few non-zero elements.
    - A very efficient structure for the diagonal matrix is to store just the entries in the main diagonal as 3 one-dimensional arrays, so a diagonal $n \times n$ matrix requires only $3n$ entries.
    - NOTE: Not all tri-diagonal matrices are sparse matrices.



Tri Diagonal Matrix

This matrix has a sparsity of 6/16 = 0.375 < 0.50 which is below the threshold to be considered a sparse matrix.

6. Are tri-diagonal matrices important? And should they impact your design? If so, how?
    - Yes, they can be important when performing some matrix operations because when operating on a tri-diagonal matrix, some operations can be processed more efficiently if the matrix is tri-diagonal. Some examples of these calculations include matrix operations such as determinants, inverses and transpositions.
    - This will impact our design as providing support for recognizing these special cases, and providing the means to create and work with tri-diagonal matrices in a sparse storage format will increase the usability, and robustness of the sparse matrix library.

7. What is a good data representation for a sparse matrix?
    - We can have substantial memory reductions by storing only the non-zero elements. It is not possible to achieve both efficient modification and efficient access time at once.
        - If we want efficient modification, we could use:
            - Dictionary of keys - Maps (Row, column) key to a value. Elements missing from the dictionary are zero. Good for constructing a sparse matrix in random order but poor for iterating over.
            - List of lists - Stores one list per **row** with each entry containing **column** index and the value. Typically sorted by column index. (The bolded words refer to the original matrix.)
            - Coordinated list - Stores a list of (row, column, value) tuples. Typically sorted by row index then column index to improve random access time.
        - If we want efficient access and matrix operations:
            - Compressed sparse row (or Yale method) - Stores a sparse matrix M in 3 arrays denoted A, JA and IA. The array A, holds all nonzero entries in M in row-major order. The array JA stores the column index in M for each element in A. Note that Length(A) == Length(JA) == # of nonzero entries. Finally, the array IA, is defined recursively:
                - IA[0] = 0
                - IA[*i*] = IA[*i* − 1] + (number of nonzero elements on the *i*-th row in the original matrix)
            - Compressed sparse column is similar to above.

8. Assume that you have a customer for your sparse matrix package. The customer states that their primary requirements as: for a N x N matrix with m non-zero entries
    - Storage should be ~O(km), where k << N and m is any arbitrary type defined in your design.
    - Adding the m+1 value into the matrix should have an execution time of ~O(p) where the execution time of all method calls in standard Ruby container classes is considered to have a unit value and p << m ideally p = 1

In this scenario, what is a good data representation for a sparse matrix?

- We should use both the “list of lists” and the new “Yale” representation. The “list of lists” implementation has a storage complexity of O(2m) which matches the above constraint for memory space. It also has an insertion time of O(1) which complies with the execution constraint of O(p). The new “Yale” implementation allows us to perform matrix operations more efficiently with a storage complexity of O((2 x NZV) + m +1) and it costs O(1) to look up a row in a Yale matrix with O(log(n)) to look up an entry within a row.

9. Design Patterns are common tricks which normally enshrine good practice
    - Explain the design patterns: Delegate and Abstract Factory
        - Delegate: This is a software design pattern where object composition is used to simulate multiple inheritances in order to take advantage of methods offered by different classes without inheriting them. The structure of the delegate pattern is as follows, Class A is composed of an object from Class B and provides an interface such that the methods from Class B can be called upon by functions in Class A.
        - Abstract Factory: This is a software design pattern that allows object creation through factory classes, without specifying what object is created and how it is created, deferring the creation behavior to be implemented inside the factory class that inherits from the abstract factory. This allows the creation of objects to be decoupled from the main program and separates the details of implementation of a set of objects from their general usage.
    - Explain how you would approach implementing these two patterns in Ruby
        - Delegate: Used when a class uses another classes’ function and uses the same name for themselves. Class A will be created in Ruby as normal, with given parameters and a method name (e.g. method1(self, parameter)). Class B will have a method also called “method1(class A object)” and return the result of calling the same method on that class A object
        - Abstract Factory: Used when we need to create multiple factory classes but also ensure that they are all using the same functions. We will create an abstract factory class that contains all general matrix functionality. Then we will create several matrix factories that inherits from our abstract factory but implements each method differently. Finally, in our sparse matrix module, we can take advantage of polymorphism to write general function calls for each matrix type.
    - Are these patterns applicable to this problem? Explain your answer! (HINT: The answer is yes)
        - The reason we require the *Delegate Design Pattern*:
            - This pattern will be helpful for implementing a common matrix class, which can operate on matrices of different storage types by

delegating the implementation of the operation to the specific implementation for that storage type. This will allow us to create a generic matrix class, and compose it with the matrix data for a specific storage type. This generic matrix class can then delegate the operation calls to the specific implementation of that operation in the storage type class that it is composed with. This design pattern is powerful here as it allows matrices with different storage types to operate on each other, regardless of the format their data is stored in.

- The reason we require the *Abstract Factory Pattern:*
  - The Abstract Factory Pattern will assist us in building matrices of specific storage types, that all follow the same build process. This allows us to provide an interface to the user to create matrices of different types, and build that data structure without exposing the implementation of how the matrix is built. This pattern also allows an efficient means to create and convert between storage types.

10. What implementation approach are you using (reuse class, modify class, inherit from class, compose with class, build new standalone class); justify your selection.
    - Reuse class: This implementation approach will be primarily used as Ruby has several linear algebra libraries. The two most important are the standard Ruby Matrix library and NMatrix. We can reuse the classes in this module to reduce the amount of code we need to write.
    - Modify class: This will be used when implementing functionality for tridiagonal matrices as most linear algebra libraries do not support this special case. This can also be used where existing libraries do not provide optimization for sparse matrices.
    - Compose with class: This will be used for the delegate design pattern to call on different matrix operations of different types
    - Inherit from class: This will be used with the abstract factory design pattern to construct our matrix factories.
    - Build New Standalone class: This will be used to provide an implementation of the “Dictionary of keys” storage type in our library, as we were unable to find ruby gems that support this storage type

11. Are iterators a good technique for sparse matrix manipulation? Are “custom” iterators required for this problem? (HINT: The answer is yes)
    - Yes, we will need custom iterators as sparse matrices are usually not stored in conventional matrix format such as a NxN array. A custom iterator would allow us to access specific elements such as, elements on the main diagonal, non-zero elements, elements in a specific column or elements in a specific row while hiding the details of the matrix storage implementation from the user.

12. What exceptions can occur during the processing of sparse matrices? And how should the system handle them?
    - Sparse Matrix is no longer sparse: This depends on the context of the situation. For example, an addition operation would not care if the output is no longer sparse, but if a non-sparse matrix is given as input this would be an issue. The solution to this will be to write a method that determines the sparsity of a matrix.
    - Matrix operations with incorrect dimensions sizes: Assertions should test dimensions before performing matrix operations and raise an exception if they do not match.
    - Matrix too large exception: Assertions should test dimensions before performing matrix operations and raise an exception if they are too large.
    - Inverting a matrix when determinant = 0: Assertions should test determinant before performing inverse operation and raise an exception if determinant = 0.
    - Matrix shape exceptions: which could be thrown when an operation is called on a non-square matrix, that requires the matrix to be square (Such as calculating the determinant). A solution to this would be to throw an error informing the user that this operation requires a square matrix.
    - Asymmetric exception: Some operations require a matrix to be symmetric to have a defined solution. A solution to this would be to check if the matrix is symmetric and if not, return a null value or throw an error.

13. What information does the system require to create a sparse matrix object? Remember you are building for a set of unknown customers – what will they want?
    - The dimensions of the matrix.
    - Different storage types of the matrix. eg) Yale, List of lists or Dense
    - Access to all elements in the matrix.
    - Factorizations of sparse matrices
    - Element-wise Operations on sparse matrices
    - Matrix operations
    - Easy to use API
    - Creation methods
    - Support for many data types

14. What are the important quality characteristics of a sparse matrix package? Reusability? Efficiency? Efficiency of what?
    - The efficiency of storage and data access, which we can achieve by ignoring zero elements.
    - The efficiency of operations on sparse matrices, which can be achieved using by performing operations on the storage types discussed. This is achieved by ignoring zero values in matrices and operating only on the indices that hold data.
    - Usability of the matrix library. There are two quality of life characteristics that will enhance the usability of our library and make code written with our library easier to read:

        - Standard naming convention of each function.
        - Verbose exception handling for better reliability
    - Reusability, Extendability, Maintainability
        - When other developers use the library, the use of consistent naming conventions will make our matrix library easier to use/work with.
    - Support for a wide range of storage types, and operations on and between these storage types

Part 3

Augmented Design Decisions Rationale:

- We added constructors for automatic creation of identity, zero, tri-diagonal and random matrices. This decision was made to increase the usability of our Sparse Matrix library as certain matrices can be constructed with less lines of code.
- We decided that optimizing each operation for tri-diagonal matrices did not complement our program design. This is because we were unable to find algorithms for tri-diagonal matrices that offered an improvement in runtime or storage space over sparse matrix implementations. Therefore, it was decided to instead support the creation and the checking of tri-diagonal matrices and to provide iterators for parsing matrix diagonals, so the user could perform their own optimizations using our library.
- We added support for “Dictionary of keys” storage type using a “build new standalone class” implementation method. This increases the scope of our library and makes it more robust.
- We added support for conversion between matrix types. This increases the usability of the library as the user has the flexibility to work with multiple matrix types.
- We discovered that the ruby gem NMatrix which we initially used heavily does not implement certain functionality that the documentation claims. As a result, we were forced to using the built-in ruby matrix functions for some of our functions.
- It decided to not implement cholesky factorization because NMatrix library did not implement the factorization despite the documentation. We instead included eigenvalue decomposition.
- Additional conditional operations were added to our set of conditionals. The conditionals added are: tridiagonal?, normal?, orthogonal?, permutation?, singular?, regular?, real?, unitary?, square?, upper_triangle?, and lower_triangle? The Hessenberg conditional operations was removed. It was decided that it was not a crucial feature to implement.

Contract Deviations:

- Additional post and pre conditions were added to our arithmetic operation contracts. This was done to improve the clarity of the operations and to demonstrate their properties.
- The Hessenberg conditional operations was removed. The corresponding contract was removed.

- Additional conditional operations were added to our set of conditionals. The conditionals added are: tridiagonal?, normal?, orthogonal?, permutation?, singular?, regular?, real?, unitary?, square?, upper_triangle?, and lower_triangle? Alongside these conditionals more contract assertions were added to each to represent their functionality and behavior. This deviation was made because we wanted to support more functionality with our library.
- Cholesky factorization was not implemented and the contract was removed.
- Eigenvalue decomposition was implemented and the corresponding contracts were added.

Additional System Testing:

- A set of test suites were created to test all operations in the SMatrix module (ie. achieve statement coverage) These tests provided an efficient way to test for operation correctness and to help identify any runtime errors that may be present in the code.
- A set of tests with randomly generated matrices (up to 100x100) was created and ran multiple times to verify operations had been implemented correctly, and unexpected errors would not arise.

Known Issues:

- Power function throws an error for LIL storage type.
- Power function throws an error for negative values (Oddly, this was observed to only happen on some of our group members computers) .
- Building a random sparse matrix is not guaranteed to be sparse. This is due to the nature of random sparse matrix generation.
- Matrix to_f only converts some values to float (some remain int for some reason, we believe the issue lies in the C++ implementation). We observed the only values not converted are zero values.
- Couldn't implement cholesky due to ruby gem.