Analysis of Step-2 code

```cpp
#include <iostream>
#include <vector>
#include <set>
#include <algorithm>
#include <random>
#include <chrono>
#include <map>
#include <unordered_map>
#include <tuple>
#include <memory>
#include <cassert>
#include <utility>
#include <functional>
#include <array>

template<typename T>
class SparseMatrix {

  public:

    using Vector = std::vector<T>;

    SparseMatrix() : _nnz(0), _nrows(0), _ncols(0) {};

    size_t getRows() const { return _nrows; };
    size_t getCols() const { return _ncols; };
    size_t getNNZ() const { return _nnz; };

    void print(std::ostream& os = std::cout) const {
      os << "nrows: " << _nrows << " | ncols:" << _ncols << " | nnz: " << _nnz << std::endl;
      _print(os);
    };

    virtual Vector vmult(const Vector& x) const = 0;

    virtual const T& operator()(size_t i, size_t j) const = 0;

    virtual T& operator()(size_t i, size_t j) = 0;

    virtual ~SparseMatrix() = default;

  protected:

    virtual void _print(std::ostream& os) const = 0;
    size_t _nnz, _nrows, _ncols;
};

template<typename T>
class MapMatrix : public SparseMatrix<T> {

  public:

    using Vector = typename SparseMatrix<T>::Vector;

    virtual Vector vmult(const Vector& x) const override {
      assert(x.size() == SparseMatrix<T>::_ncols);
      Vector res(SparseMatrix<T>::_nrows);
      for (size_t i = 0; i < _data.size(); ++i) {
        for (const auto& [j, v] : _data[i]) {
          res[i] += x[j] * v;
        }
      }
      return res;
    };

    virtual T& operator()(size_t i, size_t j) override {

      if (_data.size() < i + 1) {
        _data.resize(i + 1);
        SparseMatrix<T>::_nrows = i + 1;
      }

      const auto it = _data[i].find(j);

      if (it == _data[i].end()) {
        SparseMatrix<T>::_nnz++;
        SparseMatrix<T>::_ncols = std::max(SparseMatrix<T>::_ncols, j + 1);
        return (*_data[i].emplace(j, 0).first).second;
      } else {
        return (*it).second;
      }
    };

    virtual const T& operator()(size_t i, size_t j) const override {
      return _data.at(i).at(j);
    };

    virtual ~MapMatrix() override = default;

  protected:
    virtual void _print(std::ostream& os) const override {
      for (size_t i = 0; i < _data.size(); ++i) {
        for (const auto& [j, v] : _data[i]) {
          os << "["<< i << "," << j << "]" << " = " << v << std::endl;
        }
      }
    };

    std::vector<std::map<size_t, T>> _data;
};

int main() {
  // Implement a proper main to check if the class works as expected
}
```

What is a map of maps?

In the context of the provided C++ code, a "map of maps" is a data structure used to represent a two-dimensional sparse matrix. It is composed of nested associative data structures, where the outer structure (the "outer map") maps row indices to inner structures (the "inner maps"), which, in turn, map column indices to values.

Let's break it down:

1. **Outer Map (Row Index):** The outer map, typically an `std::map` or `std::unordered_map`, associates row indices with inner maps. Each row index maps to a specific row in the matrix.
2. **Inner Maps (Column Index to Values):** The inner maps, also `std::map` or `std::unordered_map`, are associated with individual rows. Each inner map represents a single row of the matrix, where the keys are the column indices, and the values are the non-zero elements at those positions.

This structure is ideal for representing sparse matrices, where the majority of elements are zero. It allows for efficient element access, insertion, and iteration. When you need to access an element, you look up the corresponding row and then the specific column within that row.

Here's an example of how a "map of maps" might look for a sparse matrix:

```
std::map<size_t, std::map<size_t, double>> sparseMatrix;
```

In this example:

- `sparseMatrix[0]` represents the first row and is an inner map that maps column indices to values for that row.
- `sparseMatrix[1]` represents the second row, and so on.

- Within each row, you can access elements by their column index.

For example, to access the element at row 0, column 2, you would use:

```
double value = sparseMatrix[0][2];
```

This allows for efficient access to non-zero elements in a sparse matrix without allocating memory for all possible elements. It's a memory-efficient data structure for sparse data, and the provided code uses it to implement the `MapMatrix` class for sparse matrices.

Inherited class MapMatrix: analysis

```
using Vector = typename SparseMatrix<T>::Vector;
```

The syntax `using Vector = typename SparseMatrix<T>::Vector;` in the `MapMatrix` class is used to define an alias for the `Vector` type that is inherited from the base class `SparseMatrix<T>`. This line of code essentially creates a new type alias `Vector` that refers to the `Vector` type from the base class.

Here's why this syntax is used:

1. **Inheriting a Dependent Type:** In C++, when a class inherits from a template class (in this case, `SparseMatrix<T>`), the derived class (`MapMatrix<T>`) is dependent on the template parameter `T`. This means that the derived class doesn't know about the specific types defined in the base class until it is instantiated with a concrete type.

2. **Type Aliasing:** To make the type `Vector` from the base class available in the derived class without knowing its concrete definition, a type alias is created. This alias allows the derived class to use `Vector` as if it were a local type, regardless of the template parameter `T`.

3. **`typename` Keyword:** The `typename` keyword is used because `Vector` is a dependent type, and the compiler needs to be explicitly told that it's a type. Without `typename`, the compiler would assume that `Vector` is a variable or a non-type, which would lead to a compilation error.

So, by using this syntax, the `MapMatrix` class can use `Vector` as a type, and when `MapMatrix` is instantiated with a specific type `T`, it will correctly use the `Vector` type defined in the base class `SparseMatrix<T>`. This is a common practice in C++ when dealing with dependent types in template classes.

```
virtual Vector vmult(const Vector& x) const override {

  // check if the size of the input vector is equal to the number of columns of the matrix
  assert(x.size() == SparseMatrix<T>::_ncols);
  // declare a vector of size equal to the number of rows of the matrix
  Vector res(SparseMatrix<T>::_nrows);
  for (size_t i = 0; i < _data.size(); ++i) { // iterate over the rows of the matrix
    for (const auto& [j, v] : _data[i]) {  // iterate over the non-zero elements of the curr
```

```
      res[i] += x[j] * v; // compute the contribution of the current non-zero element to th
    }
  }
  return res;
};
```

Let's break down the `MapMatrix::vmult` method step by step:

1. `assert(x.size() == SparseMatrix<T>::_ncols);`

   - This line checks whether the size of the input vector `x` is equal to the number of columns `_ncols` in the sparse matrix. It uses the `assert` function to assert that this condition is true. If the condition is false, it will trigger an assertion error, indicating that the input vector size doesn't match the number of columns.
   - The `assert` statement in C++ is used for debugging purposes. It's used to verify whether a given condition is true, and if it's not, the program will terminate with an error message. In the line `assert(x.size() == SparseMatrix<T>::_ncols);`, `assert` is written with round brackets because it's acting as a function-like macro. Here's what's happening in this line:
     - `assert(`: This starts the `assert` statement.
     - `x.size() == SparseMatrix<T>::_ncols`: This is the condition being checked. It compares the size of the vector `x` (the input vector for a matrix-vector product) with `_ncols`, which is a member variable in the `SparseMatrix` class representing the number of columns in the matrix. The purpose of this `assert` statement is to ensure that the dimensions of the input vector and the matrix are compatible for multiplication.
     - `);`: This closes the `assert` statement. It's important to note that the condition inside the `assert` statement should evaluate to true. If it evaluates to false, the `assert` macro will trigger an error, and the program will terminate, typically with an error message indicating the assertion failure.

   In this context, the `assert` statement is used for runtime debugging to catch potential issues with matrix-vector multiplication, like mismatched dimensions. If the condition is false (i.e., the dimensions don't match), the program execution will halt, helping you identify and fix the problem. Once your code is thoroughly tested, you can usually disable `assert` statements for improved performance in the release version of your software.

2. `Vector res(SparseMatrix<T>::_nrows);`

   - This line creates a new vector `res` with a size equal to the number of rows `_nrows` in the sparse matrix. This vector will store the result of the matrix-vector multiplication. It is using a C++ syntax known as direct initialization to create an instance of the `Vector` class, named `res`, and initialize it with a specified size. In this case, `Vector` is a typedef for `std::vector<T>`, so it's creating an instance of the `std::vector<T>` class. The syntax in the round brackets is essentially calling the constructor of the `std::vector<T>` class with an argument, which is the size of the vector. Here's what each part of that line does:

- `Vector res`: This declares a variable named `res` of type `Vector`, which is an alias for `std::vector<T>`. It's creating a vector object.
- `(SparseMatrix<T>::_nrows)`: Inside the round brackets, you provide an argument to the constructor. In this case, it's specifying the size of the vector. `_nrows` is an integer member variable of the `SparseMatrix` class that represents the number of rows in the matrix.

So, `Vector res(SparseMatrix<T>::_nrows);` creates a vector `res` with a size equal to the number of rows in the sparse matrix. This is a convenient way to initialize a vector with a specific size at the time of declaration.

3. `for (size_t i = 0; i < _data.size(); ++i) { ... }`

   - This loop iterates over the rows of the sparse matrix, where `_data` is a data structure that holds the non-zero elements of the matrix. It's a vector of maps, where each map represents a row of the matrix.

The `_data` member variable is a critical part of the `MapMatrix` class, and it is a data structure used to store the non-zero elements of the sparse matrix. Here's a detailed explanation of how `_data` is defined and used:

- **Definition**: `_data` is defined as a member variable within the `MapMatrix` class. It's declared as a `std::vector` of `std::map` objects. The data structure's full type is `std::vector<std::map<size_t, T>>`, where `T` is the template parameter for the `MapMatrix` class, representing the type of matrix elements (e.g., `double`)
- **Purpose**:
  - The outer `std::vector` is used to represent the rows of the sparse matrix. Each element in this vector corresponds to a row.
  - Each inner `std::map` represents a row in a compressed format. The key in the map is the column index of a non-zero element, and the associated value is the value of that element.
  - By using a map for each row, the implementation can efficiently store and look up non-zero elements by their column indices.
- **Usage**:
  - The `_data` structure is used to efficiently store non-zero elements in a way that minimizes memory consumption for sparse matrices. When a non-zero element is accessed or modified using

    the `operator()` function, it looks for the corresponding map in `_data` and accesses or updates the element in constant time.
  - In the `vmult` function, `_data` is used to iterate through non-zero elements efficiently to compute the matrix-vector product.

- **Sparse Matrix Representation**:
  - `_data` is part of a broader strategy to represent sparse matrices efficiently, known as the Compressed Sparse Row (CSR) format. In this format, only non-zero elements are stored, saving memory for matrices with many zeros.

By using this data structure, the `MapMatrix` class can store and manipulate sparse matrices in a memory-efficient way while still providing fast access to non-zero elements. It's a common approach for dealing with sparse matrices in numerical computing.

4. `for (const auto& [j, v] : _data[i]) {...}`
   - Nested within the outer loop, this loop iterates over the key-value pairs within the map representing the current row. Here, `j` is the column index, and `v` is the value at that position in the matrix.
5. `res[i] += x[j] * v;`
   - Within the inner loop, this line calculates the contribution of the current non-zero element to the result vector. It multiplies the value `v` at position `(i, j)` in the matrix by the corresponding value `x[j]` from the input vector and adds this contribution to the result vector `res[i]`. This line essentially performs the matrix-vector multiplication for the current row.
6. `return res;`
   - Finally, the method returns the result vector `res` after all rows have been processed. The result vector contains the outcome of the matrix-vector multiplication.

In summary, the `MapMatrix::vmult` method performs a matrix-vector multiplication for the sparse matrix represented by `_data`. It iterates over the rows, multiplies non-zero elements by corresponding values in the input vector `x`, and accumulates the results in the output vector `res`. The resulting vector contains the product of the sparse matrix and the input vector.

```
virtual T& operator()(size_t i, size_t j) override { // This is the write access operator

      if (_data.size() < i + 1) { // check if you have enough rows to access the row i
        // if not, resize the object to have enough rows
        _data.resize(i + 1); // resize the map of maps to have enough rows
        SparseMatrix<T>::_nrows = i + 1; // update the number of rows of the matrix
      };
      // define an iterator to the element of the map corresponding to the row i and column
      // j is the key of the map, i.e. the column index
      const auto it = _data[i].find(j); // type: std::map<size_t, T>::iterator, a pointer

      if (it == _data[i].end()) { // check if the element exists in the map
        // if not, create it
        SparseMatrix<T>::_nnz++; // update the nnz of the matrix
        SparseMatrix<T>::_ncols = std::max(SparseMatrix<T>::_ncols, j + 1); // update _ncol
        return (*_data[i].emplace(j, 0).first).second; // create the element and return a r
      } else {
        return (*it).second;
      }
};
```

Let's dive into the `operator()` method within the MapMatrix class in detail, this is the method that allows you to write non-zero elements to the matrix and read them back. Here's what's happening in this method:

1. `virtual T& operator()(size_t i, size_t j) override {...}`
   - This is the declaration of the `operator()` method. It allows you to access and manipulate the elements of the matrix by specifying the row i and column j.
2. `if (_data.size() < i + 1) { ... }`
   - This block checks whether the `_data` object has enough elements to access the row `i`. If it doesn't, it resizes the object to make sure it has enough elements to access the row. This is a common practice in C++ to ensure that a matrix has enough elements before accessing them. If you try to access an element that doesn't exist, it will lead to undefined behavior.
   - If the statement is true:
     – the `_data` object is resized to have enough elements to access the row `i`. The size of the object is set to `i + 1`, which is the index of the last element in the object. This ensures that the object has enough elements to access the row `i`, remember that C++ starts counting from 0. The complete implementation is: `_data.resize(i + 1)`. The `_data.size()` method is returning the number of rows in the sparse matrix. In the context of the provided C++ code, `_data` is a `std::vector` of `std::map` objects, where each `std::map` represents a row of the sparse matrix. So, when you call `_data.size()`, it returns the number of rows in the matrix stored in the `_data` object. It's essentially giving you the current number of elements (rows) in the object `_data`, which represents the matrix's row count. This value is used to determine whether a specific row exists or needs to be created when accessing or modifying elements in the sparse matrix;
     – the `_nrows` member variable is also updated to reflect the new

number of rows in the matrix. This is done by setting `_nrows` to `i + 1`, which is the index of the last row in the matrix. The complete implementation is: `SparseMatrix<T>::_nrows = i + 1;` and it is performed in order to match the `_nrows` member variable with the actual number of rows in the matrix. This is a common practice in C++ to ensure that the member variables of a class are consistent with the actual state of the object.

3. `const auto it = _data[i].find(j);`
   - This line defines an iterator `it` to the element of the map corresponding to the row `i` and column `j`. The iterator is used to access the element in the map and modify it if it exists. If the element doesn't exist, it will be created (see: if block).
   - The `find` method is used to find an element in the map with a specific key. In this case, the key is the column index `j`. If the element exists, the `find` method will return an iterator, i.e. a pointer, to that element. If the element doesn't exist, it will return an iterator to the end of the map. So, if the iterator `it` is equal to `_data[i].end()`, it means that the element doesn't exist in the map.
   - The `find` method is a common method in C++ maps. It's used to find an element in the map with a specific key. If the element exists, it returns an iterator to that element. If the element doesn't exist, it returns an iterator to the end of the map. This is a common practice in C++ to check whether an element exists in a map before accessing it.
4. `if (it == _data[i].end()) { ... }`
   - This block checks whether the iterator `it` is equal to `_data[i].end()`. If it is, it means that the element doesn't exist in the map, and it needs to be created. If it isn't, it means that the element exists in the map, and it can be accessed and modified.
   - If the statement is true:
     - The `_nnz` member variable is incremented to reflect the addition of a new non-zero element to the matrix. This is done by setting `_nnz` to `_nnz + 1`, which is the current number of non-zero elements in the matrix plus one. The complete implementation is: `SparseMatrix<T>::_nnz++;` and it is performed to match the `_nnz` member variable with the actual number of non-zero elements in the matrix. This is a common practice in C++ to ensure that the member variables of a class are consistent with the actual state of the object.
     - The `_ncols` member variable is updated to reflect the new number of columns in the matrix. This is done by setting `_ncols` to the maximum between `_ncols` and `j + 1`, where `j` is the column index of the new element. The complete implementation is: `SparseMatrix<T>::_ncols = std::max(SparseMatrix<T>::_ncols, j + 1);` and it is performed to match the `_ncols` member variable with the actual

number of columns in the matrix. This is a common practice in C++ to ensure that the member variables of a class are consistent with the actual state of the object.

  - The new element is created in the map using the `emplace` method. This method creates a new element in the map with a specific key and value. In this case, the key is the column index `j`, and the value is `0`, just for initializing the value. The complete implementation is: `_data[i].emplace(j, 0)`. This method returns a pair of iterators, where the first iterator points to the newly created element in the map. The second iterator points to a boolean value indicating whether the insertion was successful. If the key didn't exist in the map and a new element was inserted, `.second` will be `true`. If the key already existed, it will be `false`. In this case, the second iterator is ignored, and only the first iterator is used. The first iterator is dereferenced to access the newly created element, and the `.second` member variable is used to access the value of that element. The complete implementation is: `(*_data[i].emplace(j, 0).first).second;`. This line returns a reference to the newly created element in the map, which is then used to assign a value to that element. This is a common practice in C++ to create a new element in a map and assign a value to it.
  - If the statement is false: the element already exists in the map, and it can be accessed and modified. In this case, the `operator()` method returns a reference to the existing element in the map. This is a common practice in C++ to access an element in a map. The complete implementation is: `return (*it).second;`. This line returns a reference to the existing element in the map, which is then used to assign a value to that element.

```
virtual const T& operator()(size_t i, size_t j) const override {
  // same as _data[i][j] or _data[i].at(j); the one employing at() is safer
  return _data.at(i).at(j);
};
```

Let's break down this `operator()` method step by step, this is the method that allows you to read non-zero elements from the matrix. Here's what's happening in this method:

1. `virtual const T& operator()(size_t i, size_t j) const override`: This is the declaration of the operator method. It's marked as `const` because it doesn't modify the state of the object it's called on. It takes two arguments, `i` and `j`, which represent row and column indices. The return type is a constant reference to a value of type `T`.

2. `return _data.at(i).at(j);`: This line returns an element from a 2D data structure stored in the `_data` member of the class.

- `_data`: This is a private member of the class that holds the sparse matrix data. It's a vector of maps (`std::vector<std::map<size_t, T>>`), where each map represents a row of the sparse matrix. The outer vector represents rows, and each map inside it stores non-zero values for the corresponding row.
- `.at(i)`: This part accesses the `i`-th element of the `_data` vector. It corresponds to the `i`-th row of the sparse matrix.
- `.at(j)`: Inside the `i`-th row (the map), this part accesses the `j`-th element, which corresponds to the value at row `i` and column `j`.
- The whole expression retrieves the element at the specified `i` and `j` indices in the sparse matrix data structure and returns it as a constant reference to a value of type `T`.

The reason for using `.at(i).at(j)` instead of `_data[i][j]` or `_data[i].at(j)` is mainly for safety. Here's why:

1. Safety: The `std::map::at` function performs bounds checking, ensuring that you don't access elements that are out of bounds. If the specified indices `i` and `j` are invalid (e.g., out of bounds), it will throw an exception (`std::out_of_range`), preventing undefined behavior. This provides more robust error handling compared to the `operator[]`, which doesn't provide such checks and can lead to undefined behavior if used with invalid indices.
2. Consistency: Using `.at(i).at(j)` consistently throughout the codebase ensures that bounds checking is performed for all access to the sparse matrix data. This makes the code safer and easier to reason about.

So, by using the `.at(i).at(j)` approach, the code prioritizes safety and robustness, making it a good practice for handling sparse matrix data, especially in scenarios where invalid indices could be a concern.