Heat Equation Program Report

by

Matteo Barbera | 4438590
Sam van Elsloo | 4487060

Supervisor - Dr. M. Möller

Delft University of Technology

Contents

1
REPORT

This report gives an overview of the code written to approximate the solution of the heat equation, provided in the Appendix. The code was tested using the `<assert>` library, with the assert calls placed inside a preprocessor `ifndef` block inside the `main()` function. Each part of the code is briefly explained and a verification section referring to the tests used in the aforementioned `ifndef` block follows directly after every explanation.

1.1. VECTOR CLASS: CONSTRUCTORS

The Vector class contains two member variables, an integer containing information on the size of the vector and a pointer to the array of data of the vector. The type of this pointer is determined upon initialisation through a template parameter. It has a standard constructor initialising the size and pointer to 0 and `nullptr` respectively. The class can be further initialised by specifying the length of the vector, or with an initializer list. The former is an explicit constructor to prevent confusing initialisation of the Vector by a single element or by giving it a length value, and will create a pointer to an array of length specified by the input value. The latter uses the `<memory>` library to transfer the data and information on the length to the initialized object.

The class also has a standard implementation of the copy/move constructors and assignment operations, making use of `const` references to Vector objects as much as possible to prevent having to pass the potentially long data arrays by value. `const` was obviously not applicable for the move operations.

1.1.1. VERIFICATION

The initialisation of the Vector class was tested by initialising a Vector object first with an initializer list, after which an assert call was made to check that one of its elements was equal to the corresponding element in the initializer list, and second by giving the Vector class with a length value, with an assert call checking that its length really was what was specified.

1.2. VECTOR CLASS: OPERATORS

The addition and subtraction operators for the Vector class were overloaded to enable operations between two Vector objects. The + operator was overloaded for the case that the element that follows it is a Vector object, passed as a `const` reference to avoid passing the potentially larger objects by value. The operator returns a new Vector object with the result of the element-wise addition, and the type of its data is determined via a `decltype` call on the result of the addition of the first element of the two Vectors, to allow for the proper addition of Vectors with different data types. This overloaded operator was declared to be a constant member function to ensure that the content of the Vector remains unchanged. The subtraction operator works in the same way.

A definition for the += and -= operator was also written to allow the element-wise increase and decrease by another Vector object. These member functions could not be defined as constant due to the nature of the operators. All of the operators described so far have an `if` statement before the operation to check that the length of the two Vectors involved is the same, as the operations cannot be carried out on Vectors with a different number of elements.

The multiplication operator was overloaded to allow multiplication of each element of the Vector by a scalar. As both left and right multiplication had to be supported, the definition for left multiplication had to be written outside of the class definition for it to work. Again, these operators return a new Vector object with the data type determined with a `decltype` call. The member function version was specified as `const` to prevent undesired modification of the elements of the Vector calling the operator.

The last operator was not one specified by the assignment, but its addition allowed for more elegant and less cluttered code. The [] operator was overloaded to bypass having to access the data member variable to call one of its elements. A constant and non-constant version was created so that the compiler could ensure that the `const` clause is upheld whenever the operator is used in another `const` function.

1.2.1. VERIFICATION

Short Vector objects with known elements were created, and every operator was tested by using assert calls to make sure that the result of the operation was correct. The simple calculations were carried out by hand and compared with the new or modified Vector object.

1.3. VECTOR CLASS: ADDITIONAL MEMBER FUNCTIONS

Two extra member functions were added to the Vector class. The first was a function that would return the sum of all the elements in the Vector, using the Kahan sum algorithm. This member function was never used. The other addition was a constant `void` function that would simply display the Vector and its elements in an ordered fashion in the standard output stream. This member function was mostly used to visually inspect the Vectors to quickly find obvious bugs, but was seen as important for normal use as well.

1.3.1. VERIFICATION

The sum function was tested by applying it to a short Vector and comparing it to the solution computed by hand. The show function was tested by visual inspection of the output to the console.

1.4. DOT FUNCTION

This function was written to enable the dot product operation between two Vectors. A short `if` statement first checks that the two Vectors, passed as constant references due to their size and because they themselves will not be modified, have the same length, as the dot product cannot be carried out on Vectors with a different number of elements. The current implementation only supports the dot product of two Vectors with the same data type, although future work on the code would involve adding support for different data types.

1.4.1. VERIFICATION

The dot product function was tested by applying it to two known Vector objects, and the result was compared with the correct value, previously computed by hand, with an assert call.

1.5. MATRIX CLASS: CONSTRUCTORS

The Matrix class has three member variables, two integers containing information on the length of the rows and columns, and a `map` object, from the standard library `<map>`. All of the variables are private, as it should not be possible to change the size of the matrix after initialisation.

There were a number of reasons to store the data in a `map` object as opposed to a Vector. The object assigns a unique key to every value, allowing the index of each Matrix element to be elegantly specified by an `array` from the `<array>` standard library. Moreover, the element lookup speed is logarithmic, and allows range based `for` loops over the elements. Due to the `map` object being defined in another class, the constructors for the Matrix class are quite basic, as they only need to take care of initialising the row and column integers. The only unusual implementation is that the default constructor was suppressed, since there would be no point in initialising a Matrix object with no rows or columns that cannot be changed afterwards due to the variables being private.

1.5.1. VERIFICATION

The initialisation of the Matrix class was tested by creating a small Matrix object and inspecting it visually with the `show()` function, described in a further section.

1.6. MATRIX CLASS: OPERATORS

The only operator that was overloaded for the Matrix class was the [] operator. Two versions were defined, a constant and a non-constant version, that call the correct matrix element from the `map` object. The constant version uses the `map` member function `at()`, as the [] operator of the `map` object returns a modifiable reference. An `if` statement in the definition of the operator checks that the element being called is within the Matrix dimensions, as the `map` object does not have any knowledge of how it is being used.

1.6.1. VERIFICATION

The [] operator was tested by calling a known element of a Matrix object and comparing the value to what it should be.

1.7. MATRIX CLASS: MEMBER FUNCTIONS

The Matrix class has two member functions, one called `show()` that works very much like the Vector's version, and the `matvec()` function. The `show()` function outputs the values of the matrix in an ordered fashion in the standard output stream, but it had to be written with a lot of `try`/`catch` blocks as not all elements are guaranteed to be specified, in which the `map` object would return an error. Moreover, the values are called using `at()`, as calling a value, whether it is then specified or not, using [] would allocate memory for it in the object. With this implementation the sparse property of the Matrix is maintained, and the code is more memory efficient.

The `matvec()` allows for a matrix-vector multiplication to be carried out. The function checks that the Matrix and Vector dimensions are compatible before iterating over the elements of the `map` object using a range based `for` loop. The correct index of both Vectors in the loop is determined from the key of the `map` element being used. The Vector elements of the output are first initialised to 0 in a `for` loop over its elements, to avoid adding a value to an uninitialised variable, which would behave unpredictably. This implementation allows for only computing the product between the Matrix and Vector that would result in a non-zero answer, greatly speeding up the routine, especially for large, sparse matrices.

1.7.1. VERIFICATION

The `show()` function was tested by visual inspection of its output to the console. The `matvec()` function was tested by executing the matrix-vector product of a small Matrix and Vector with known elements, and the result compared with the correct value, which was computed manually.

1.8. CONJUGATE GRADIENT FUNCTION

The `cg` function was written by translating the pseudo-code explained in the assignment description into actual code, with the only difference being that Vector object were reused as much as possible to reduce the impact of the routine on the memory of the computer.

1.8.1. VERIFICATION

The `cg` function was tested by taking a known case with a solution described in the Wikipedia page of the Conjugate Gradient Method and comparing it with the output generated by the program.

1.9. HEAT1D CLASS

The `Heat1D` class has five member variables, three `doubles` to store the α coefficient, the time step and the spatial distance between nodes, an `int` with the number of nodes being used, and a Matrix object to store the coefficients resulting from the second order central finite difference method being used. The class only has one constructor that allows initialisation, which stores the values of α, time step and the number of nodes, determines the spatial step, and builds the coefficient Matrix. Only the non-zero values are stored during this process to save memory, as most of the elements are 0. All of the member variables are private, as they are not expected to change and should not be changed in an uncontrolled manner to maintain the validity of the solution.

The default constructor and all the copy/move assignment and operators are suppressed, as for the purpose of the code there is no scenario where they would be used, and so even the defaults are suppressed to prevent unwanted behaviour.

The class has three member functions, the first outputting a Vector containing the exact solution calculated from the analytical formulation of the answer, the second calculates the numerical approximation of the solution, and the third calls the `show()` function of the Matrix object, which is otherwise inaccessible due to the variable being private. All of the member functions are constant as they are not meant to affect the value of the class's member variables.

1.9.1. VERIFICATION

The construction of the coefficient matrix was tested by visually comparing the output of the program for a specific value of α, nodes, and time step, with the known solution given in the assignment description. The verification and validation of the numerical solution is explained at the end of this report.

1.10. HEAT2D CLASS

The structure and implementation of the 2D version of the `Heat` class is very similar to what was explained for the `Heat1D` version, the only difference being the extra `if` statements used to construct the coefficient Matrix. The verification of this class is exactly the same as that carried out for `Heat1D`.

1.11. INITIAL VALUE VECTOR

The initial value vector $\mathbf{u}(\mathbf{x},0)$ had to be computed. The initial condition was given to

$$u(\mathbf{x},0) = \prod_{k=0}^{n-1} \sin(\pi x_k) \qquad \forall \mathbf{x} \in \Omega \tag{1.1}$$

This function must be evaluated for all nodes in the domain. This is mostly just a matter of finding the coordinates corresponding to the nodes efficiently. This can be done straightforwardly. Consider a vector $\mathbf{a}$ containing the location of the nodes within a single dimension, i.e. it is given by $\mathbf{a} = [a_0, a_1, ..., a_{m-2}, a_{m-1}]^T = [\Delta x, 2\Delta x, ..., (m-1)\Delta x, m\Delta x]^T$, which has m entries. Now, as an example, consider a $10 \times 10 \times 10$ domain, containing 1000 nodes in total. A node with index 725 then would then correspond to a point of which the x_1-coordinate equals a_4 (the fifth entry of a would be considered, which is a_4), the x_2-coordinate equals a_1 and the x_3-coordinate equals a_6. In other words, for the first dimension the index is first divided by 10^0 and rounded down, after which the modulo of 10 is taken; for the second dimension, the index is divided by 10^1 and rounded down, after which the modulo of 10 was taken; for the third dimension, the index is divided by 10^2 and rounded down, after which the modulo of 10 was taken.

Indeed, the algorithm shown in Algorithm 1 can be used to set up the initial condition.

The implementation of this is shown in lines 492-507 of the code shown in Listing A.1.

1.11.1. VERIFICATION

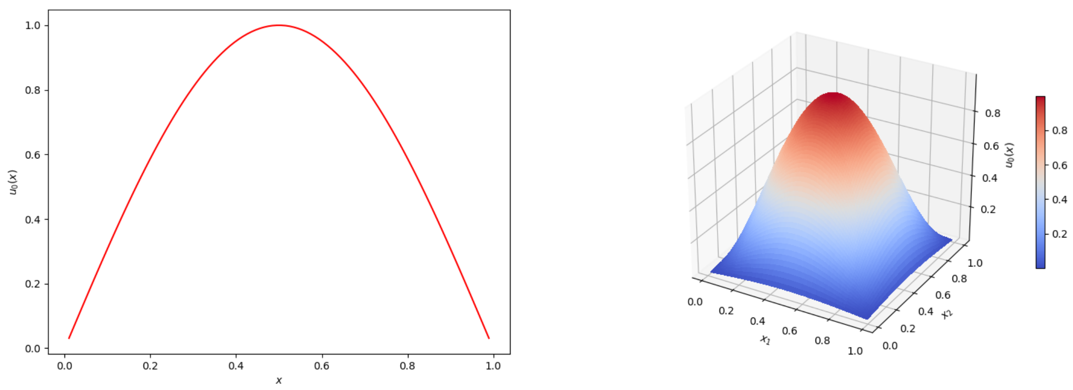
This part of the code was verified by plotting the results of the initial value vector in Python for the 1D and 2D case; this has been shown in Figure 1.1. Both plots show the expected behaviour in how Equation (1.1)

Algorithm 1 Algorithm for computing the initial value vector.

Require: number of dimensions n, number of entries per dimension m
  $\Delta x = 1/(m+1)$
  create a vector **a** of length m
  for $i = 0$ to $i = m-1$ **do**
    $a[i] = i \cdot \Delta x$
  end for
  create m^n dimensional vector $\mathbf{u}_0$ with every entry equalling 1
  for $i = 0$ to $i = m^n - 1$ **do**
    for $j = 0$ to $j = n-1$ **do**
      $k = \lceil x/m^j \rceil \mod m$
      $\mathbf{u}_0[i] = \mathbf{u}_0[i] \cdot \sin(\pi \cdot a[k])$
    end for
  end for

---

should look like, establishing the correct implementation of the algorithm. Additional tests were performed by increasing the mesh refinement; this lead to the same plots, meaning the algorithm is indeed consistent.



(a) Plot of the initial condition in 1D ($n = 1$ and $m = 100$). (b) Plot of the initial condition in 2D ($n = 2$ and $m = 100$).

Figure 1.1: Plot of the initial conditions in 1D and 2D.

1.12. RMSE FUNCTION

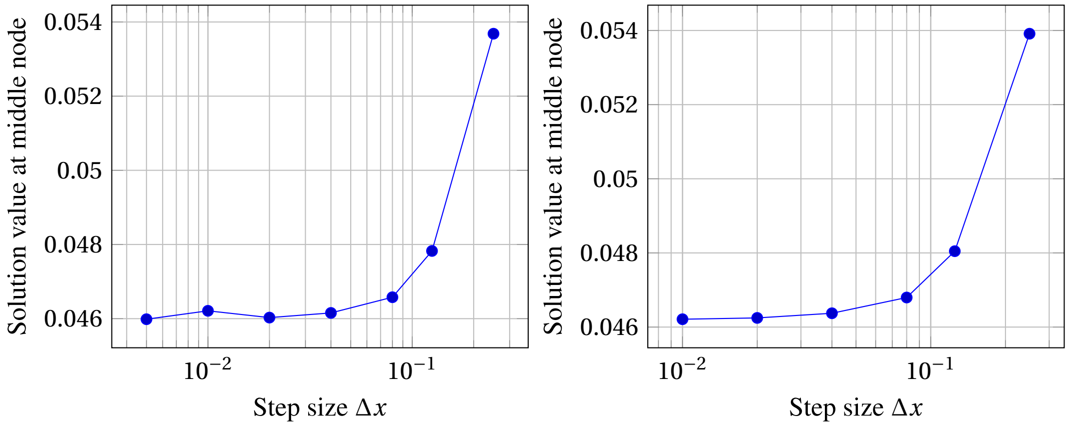
This function, even though it was not specified in the assignment, was added in order to compare the numerical with the exact solution. The function calculates and returns the root mean square error between two Vector objects, i.e. given two vectors **a** and **b** containing elements $a_1, ..., a_n$ and $b_1, ..., b_n$, it computes

$$\text{RMSE} = \sqrt{\frac{\sum_i^n (a_i - b_i)^2}{n}}$$

1.13. VERIFICATION OF FINAL CODE

First, the spatial discretisation was verified. The timestep dt was kept fixed at $dt = 0.001$, and the mesh size m was varied, resulting in various step sizes Δx. The results are plotted in Figure 1.3; evidently, the error decreases as the step sizes decreases, confirming consistency of the spatial discretisation.

Secondly, the time stepping was verified. The mesh size m was now kept fixed at $m = 99$, and the timestep dt was varied. The value at the middle point of the domain, i.e. $x_1 = 0.5$ for the 1D case and $(x_1, x_2) = (0.5, 0.5)$,

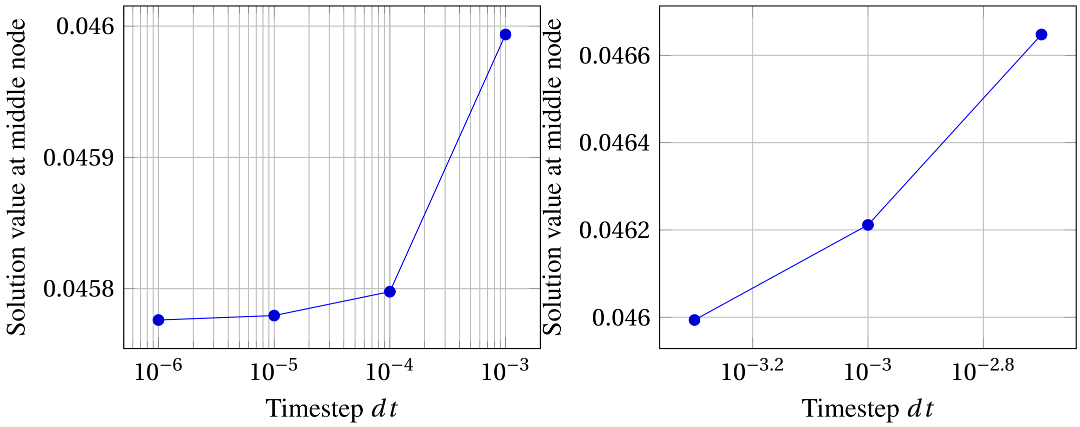


(a) Value of the solution at $x_1 = 0.5$ for the 1-dimensional case, for various mesh sizes after $t = 1$, with $dt = 0.001$.

(b) Value of the solution at $(x_1, x_2) = (0.5, 0.5)$ for the 2-dimensional case, for various time-step sizes after $t = 0.5$, with $dt = 0.001$.

Figure 1.2: Plot of the solution value at the middle point in the domain after some period of time with varying mesh sizes, with $dt = 0.001$.

after $t = 1$ and $t = 0.5$ respectively, were taken and plotted in Figure 1.3. As apparent, both the 1D and 2D case converge in stable manner, confirming the consistency of the timestepping.



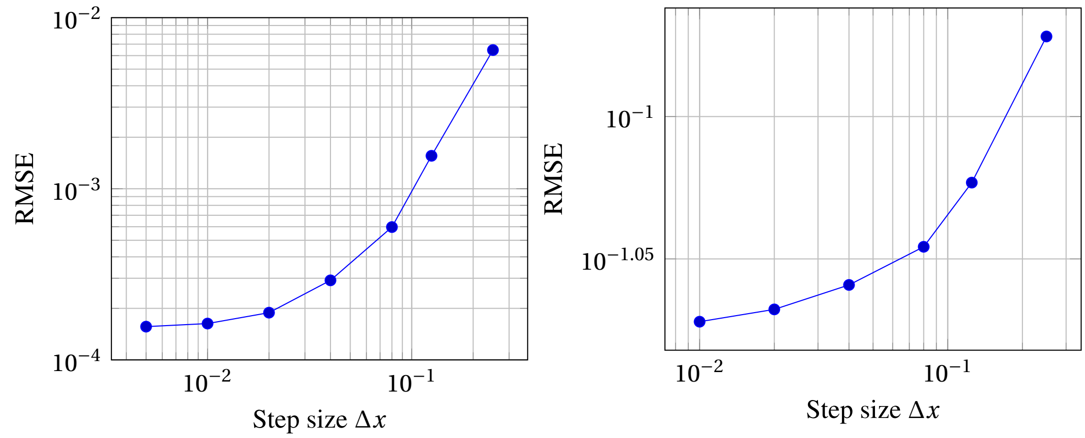
(a) Value of the solution at $x_1 = 0.5$ for the 1-dimensional case, for various time-step sizes after $t = 1$, with $m = 99$.

(b) Value of the solution at $(x_1, x_2) = (0.5, 0.5)$ for the 2-dimensional case, for various time-step sizes after $t = 0.5$, with $m = 99$.

Figure 1.3: Plot of the solution value at the middle point in the domain after some period of time with varying time-step sizes, with $m = 99$.

1.14. VALIDATION OF FINAL CODE

Now that the program was verified, it could be validated with the exact solution. To do so, the RMSE was computed for various mesh sizes with a fixed timestep of $dt = 0.001$ after $t = 1$ for the 1D case and $t = 0.5$ for the 2D case. The RMSE for various mesh sizes has been shown in Figure 1.4. Evidently, the RMSE decreases with decreasing step size, validating the model for both the 1D and 2D case.



(a) RMSE for the 1-dimensional case, for various mesh sizes after $t = 1$, with $dt = 0.001$.

(b) RMSE for the 2-dimensional case, for various time-step sizes after $t = 0.5$, with $dt = 0.001$.

Figure 1.4: Plot of the RMSE after some period of time with varying mesh sizes, with $dt = 0.001$.

A

PROGRAM CODE

```
#include <iostream>
#include <initializer_list>
#include <memory>
#include <typeinfo>
#include <map>
#include <array>
#include <cmath>
#include <assert.h>
#include <stdexcept>
// #define NDEBUG

template<typename T>
class Vector
{
public:
  int length;
  T* data;

  // Default constsructor
  Vector()
  : length(0),
    data(nullptr)
  { }

  // Length constructor
  explicit Vector(int length)
  : length(length),
    data(new T[length])
  { }

  // Initializer list constructor
  Vector(const std::initializer_list<T>& l)
  : Vector((int)l.size())
  {
    std::uninitialized_copy(l.begin(), l.end(), data);
  }

  // Copy constructor
  Vector(const Vector<T>& V)
  : length(V.length),
    data(new T[V.length])
  {
    for (int i = 0; i < length; i++)
      data[i] = V[i];
  }

  // Move constructor
  Vector(Vector<T>&& V)
  : length(V.length),
    data(V.data)
  {
   V.data = nullptr;
   V.length = 0;
  }

  // Destructor
```

```
~Vector()
{
  delete[] data;
  length = 0;
}

// Copy assignment
Vector<T>& operator=(const Vector<T>& V)
{
  T* tmp = new T[V.length];
  for (int i = 0; i < V.length; i++)
    tmp[i] = V[i];

  delete[] data;
  data = tmp;
  length = V.length;

  return *this;
}

// Move assignment
Vector<T>& operator=(Vector<T>&& V)
{
  delete[] data;
  data = V.data;
  length = V.length;
  V.data = nullptr;
  V.length = 0;

  return *this;
}

// Addition with other Vector
template <typename U>
auto operator+(const Vector<U>& V) const
{
  if (length != V.length) throw std::length_error("Cannot add Vectors of different lengths");

  Vector<decltype(data[0] + V[0])> res(V.length);
  for (int i = 0; i < V.length; i++)
    res[i] = data[i] + V[i];
  return res;
}

// Subtraction with other Vector
template <typename U>
auto operator-(const Vector<U>& V) const
{
  if (length != V.length) throw std::length_error("Cannot subtract Vectors of different lengths");

  Vector<decltype(data[0] - V[0])> res(V.length);
  for (int i = 0; i < V.length; i++)
    res[i] = data[i] - V[i];
  return res;
}

// Increase by other Vector
Vector<T>& operator+=(const Vector<T>& V)
{
  if (length != V.length) throw std::length_error("Cannot add Vectors of different lengths");

  for (int i = 0; i < V.length; i++)
    data[i] += V[i];
  return *this;
}

// Decrease by other Vector
Vector<T>& operator-=(const Vector<T>& V)
```

```
  {
    if (length != V.length) throw std::length_error("Cannot add Vectors of different lengths");

    for (int i = 0; i < V.length; i++)
      data[i] -= V[i];
    return *this;
  }

  // Return Vector entry i (modifiable)
  T& operator[](const int indx)
  {
    if (indx >= length) throw std::out_of_range("Index out of bounds");

    return data[indx];
  }

  // Return Vector entry i (constant)
  const T& operator[](const int indx) const
  {
    if (indx >= length) throw std::out_of_range("Index out of bounds");

    return data[indx];
  }

  // Multiplication by scalar on RHS
  template<typename S>
  auto operator*(S val) const
  {
    Vector<decltype(data[0]*val)> res(length);
    for (int i = 0; i < length; i++)
      res[i] = data[i]*val;

    return res;
  }

  // Return sum of elements (Kahan sum algorithm)
  T sum() const
  {
    T sum = 0;
    T c = 0;
    T y, t;
    for (int i=0; i < length; i++)
    {
      y = data[i] - c;
      t = sum + c;
      c = (t - sum) - y;
      sum = t;
    }
    return sum;
  }

  // Print Vector elements
  void show() const
  {
    std::cout << "[";
    for (int i=0; i < length - 1; i++)
      std::cout << data[i] << ", ";
    std::cout << data[length - 1] << "]" << std::endl;
  }
};

// Multiplication by scalar on LHS
template <typename S, typename T>
auto operator*(S val, const Vector<T>& V)
{
  Vector<decltype(V[0]*val)> res(V.length);
  for (int i=0; i < V.length; i++)
    res[i] = V[i]*val;
  return res;
}
```

```cpp

// Dot product between two Vectors
template <typename T>
T dot(const Vector<T>& l, const Vector<T>& r)
{
  if (l.length != r.length) throw std::length_error("Cannot find dot products of Vectors of
    different lengths");

  T res = 0;
  for (int i=0; i < l.length; i++)
    res += l[i]*r[i];
  return res;
}

template <typename T>
class Matrix
{
  int rows;
  int cols;
  std::map<std::array<int,2>, T> data;
public:
  // No default constructor
  Matrix()=delete;

  // Rows columns constructor
  Matrix(int rows, int cols)
  : rows(rows),
    cols(cols)
  { }

  // Copy constructor
  Matrix(const Matrix<T>& M)
  : rows(M.rows),
    cols(M.cols),
    data(M.data)
  { }

  // Move constructor
  Matrix(Matrix<T>&& M)
  : rows(M.rows),
    cols(M.cols),
    data(std::move(M.data))
  {
    M.rows = 0;
    M.cols = 0;
  }

  ~Matrix()
  {
    rows = 0;
    cols = 0;
  }

  // Copy assignment
  Matrix<T>& operator=(const Matrix<T>& M)
  {
    data = M.data;
    rows = M.rows;
    cols = M.cols;

    return *this;
  }

  // Move assignment
  Matrix<T>& operator=(Matrix<T>&& M)
  {
    data = M.data;
    rows = M.rows;
    cols = M.cols;
    M.data.erase;
  }
```

```

// Return Matrix entry i,j (modifiable)
T& operator[](const std::array<int, 2>& indx)
{
  if (indx[0] >= rows || indx[1] >= cols) throw std::out_of_range("Index out of bounds");

  return data[indx];
}

// Return Matrix entry i,j (constant)
const T& operator[](const std::array<int, 2>& indx) const
{
  if (indx[0] >= rows || indx[1] >= cols) throw std::out_of_range("Index out of bounds");

  T entry;
  try {
    entry = data.at(indx);
  } catch(const std::out_of_range&) {
    entry = 0;
  }
  return entry;
}

// Matrix-Vector product
Vector<T> matvec(const Vector<T>& V) const
{
  if (cols != V.length) throw std::length_error("Matrix rows and Vector length need to match");

  Vector<T> res(rows);
  for (int i=0; i < rows; i++)
    res.data[i] = 0;

  for (auto it = data.cbegin(); it != data.cend(); ++it)
  {
    res.data[(*it).first[0]] += (*it).second * V.data[(*it).first[1]];
  }
  return res;
}

// Print Matrix elements
void show() const
{
  std::cout << "[";
  for (int i=0; i < rows - 1; i++)
  {
    std::cout << "[";
    for (int j=0; j < cols-1; j++)
    {
      try {
        std::cout << data.at({i,j}) << ", ";
      } catch(const std::out_of_range&) {
        std::cout << 0 << ", ";
      }
    }
    try {
      std::cout << data.at({i, cols-1}) << "]\n ";
    } catch(const std::out_of_range&) {
      std::cout << 0 << "]\n ";
    }
  }
  std::cout << "[";
  for (int j=0; j < cols-1; j++)
  {
    try {
      std::cout << data.at({rows-1,j}) << ", ";
    } catch(const std::out_of_range&) {
      std::cout << 0 << ", ";
    }
  }
  try {
```

```
      std::cout << data.at({rows-1,cols-1}) << "]]" << std::endl;
    } catch(const std::out_of_range&) {
      std::cout << 0 << "]]" << std::endl;
    }
  }
};

// Conjugate gradient function
template <typename T>
int cg(const Matrix<T>& A, const Vector<T>& b, Vector<T>& x_k, const T tol, const int maxiter
    )
{
  Vector<T> p_k = b - A.matvec(x_k);
  Vector<T> r_k = p_k;
  T alpha, beta, dot_rk, dot_rknew;
  int iter_count = 0;
  for (int k=0; k < maxiter; k++)
  {
    dot_rk = dot(r_k, r_k);  // so you can ow r_k
    Vector<T> Ap_k(A.matvec(p_k));
    alpha = dot_rk / dot(p_k, Ap_k);
    x_k += alpha * p_k;  // save mem
    iter_count += 1;
    r_k -= alpha * Ap_k;
    dot_rknew = dot(r_k, r_k);
    if (sqrt(dot_rknew) < tol)
      return iter_count;
    beta = dot_rknew / dot_rk;
    p_k = r_k + beta * p_k;
  }
  return -1;
}

class Heat1D
{
  double alpha;
  int m;
  double dx;
  double dt;
  Matrix<double> M_iter;
public:

  Heat1D()=delete;

  Heat1D(double alpha, int m, double dt)
  : alpha(alpha),
    m(m),
    dx(1./(m + 1)),
    dt(dt),
    M_iter(Matrix<double>(m,m))
  {
    std::cout << "Setting up matrix..." << std::endl;
    for (int i=0; i < m; i++)
    {
      for (int j=0; j < m; j++)
      {
        if (i == j)
          M_iter[{i,j}] = 1 - alpha * (dt / (dx*dx))*-2;
        else if (abs(i - j) == 1)
          M_iter[{i,j}] = -alpha * (dt / (dx*dx));
      }
    }
    std::cout << "Done" << std::endl;
  }

  // Suppress copy/move operations
  Heat1D(const Heat1D&)=delete;
  Heat1D& operator=(const Heat1D&)=delete;
  Heat1D(Heat1D&&)=delete;
  Heat1D& operator=(Heat1D&&)=delete;
```

```
  // Destructor
  ~Heat1D()
  {
    alpha = 0;
    m = 0;
    dx = 0;
    dt = 0;
  }

  // Return exact solution at t
  Vector<double> exact(const double t, const Vector<double>& u_0) const
  {
    return exp(-M_PI*M_PI*alpha*t) * u_0;
  }

  // Return numerical solution
  Vector<double> solve(const double t_end, const Vector<double>& u_0) const
  {
    std::cout << "Solving to t_end = " << t_end << std::endl;
    Vector<double> res(m);
    for (int i=0; i < res.length; i++)
      res[i] = 0;
    int steps = (int)(t_end / dt);
    Vector<double> u_old(u_0);
    int maxiter(5);
    double tol(1e-08);
    for (int i=0; i < steps; i++)
    {
      cg(M_iter, u_old, res, tol, maxiter);
      u_old = res;
    }
    std::cout << "Done" << std::endl;
    return res;
  }

  // Print coefficient matrix
  void show_mat() const { M_iter.show(); }
};

class Heat2D
{
  double alpha;
  int m;
  double dx;
  double dt;
  Matrix<double> M_iter;
public:

  Heat2D()=delete;

  Heat2D(double alpha, int m, double dt)
  : alpha(alpha),
    m(m),
    dx(1./(m + 1)),
    dt(dt),
    M_iter(Matrix<double>(m*m,m*m))
  {
    std::cout << "Setting up matrix..." << std::endl;
    for (int i=0; i < m*m; i++)
    {
      for (int j=0; j < m*m; j++)
      {
        if (i == j)
          M_iter[{i,j}] = 1 - alpha * (dt / (dx*dx))*(-4);
        else if (abs(i - j) == m)
          M_iter[{i,j}] = -alpha * (dt / (dx*dx));
        else if (i - j == 1 && i % m != 0)
          M_iter[{i,j}] = -alpha * (dt / (dx*dx));
        else if (j - i == 1 && j % m != 0)
          M_iter[{i,j}] = -alpha * (dt / (dx*dx));
      }
```

```
      }
      std::cout << "Done" <<std::endl;
    }

  // Suppress copy/move operations
  Heat2D(const Heat1D&)=delete;
  Heat2D& operator=(const Heat1D&)=delete;
  Heat2D(Heat1D&&)=delete;
  Heat2D& operator=(Heat1D&&)=delete;

  // Destructor
  ~Heat2D()
  {
    alpha = 0;
    m = 0;
    dx = 0;
    dt = 0;
  }

  // Return exact solution at t
  Vector<double> exact(const double t, const Vector<double>& u_0) const
  {
    return exp(-M_PI*M_PI*alpha*t) * u_0;
  }

  // Return numerical solution
  Vector<double> solve(const double t_end, const Vector<double>& u_0) const
  {
    std::cout << "Solving to t_end = " << t_end << std::endl;
    Vector<double> res(m*m);
    for (int i=0; i < res.length; i++)
      res[i] = 0;
    int steps = (int)(t_end / dt);
    Vector<double> u_old(u_0);
    int maxiter(5);
    double tol(1e-08);
    for (int i=0; i < steps; i++)
    {
      cg(M_iter, u_old, res, tol, maxiter);
      u_old = res;
    }
    std::cout << "Done" << std::endl;
    return res;
  }

  // Print coefficient matrix
  void show_mat() const { M_iter.show(); }
};

// Generate initial value of solution at time t=0
Vector<double> u_init(const int m, const int n)
{
  double dx = 1./(m + 1);
  Vector<double> x(m);
  for (int i=0; i < x.length; i++)
    x[i] = (i+1)*dx;

  double mn = pow(m, n);
  Vector<double> u_0(mn);
  for (int i=0; i < u_0.length; i++)
    u_0[i] = 1.;

  for (int i=0; i < (int)mn; i++)
    for (int j=0; j < n; j++)
      u_0[i] *= sin(M_PI*x[(int)(i/pow(m,j))%m]);
  return u_0;
}

// Compute RMS of a Vector
template <typename T>
T RMS(const Vector<T>& vec)
```

```
{
  T rmse = 0;
  for (int i=0; i < vec.length; i++)
    rmse += vec[i]*vec[i];
  return sqrt(rmse / vec.length);
}

int main()
{
// Tests
#ifndef NDEBUG
  // Tolerance for float comparison
  double eps = 1e-07;

  Vector<int> a({1,2,3,4});
  assert (a[1] == 2);
  Vector<double> a2(4);
  assert (a2.length == 4);
  for (int i=1; i!=a2.length+1; i++)
    a2[i-1] = 2.0 * i;

  // scalar on the right
  auto b = a*2.1;
  assert (typeid(b[0]) == typeid(double));
  assert (abs(b[1] - 4.2) < eps);
  auto b2 = a * 2;
  assert (typeid(b2[0]) == typeid(int));
  assert (b2[0] == 2);
  // scalar on the left
  auto c = 2.1*a;
  assert (typeid(c[0]) == typeid(double));
  assert (abs(c[1] - 4.2) < eps);
  auto c2 = 2*a;
  assert (typeid(c2[0]) == typeid(int));
  assert (c2[0] == 2);
  // + - operators
  auto d = a - a2;
  assert (typeid(d[0]) == typeid(double));
  assert (abs(d[1] + 2) < eps);
  auto d2 = a + a2;
  assert (abs(d2[1] - 6) < eps);
  // dot product
  auto e = dot(a, a);
  assert (abs(e - 30) < eps);

  Matrix<double> M(3,2);
  for (int i=0; i < 3; i++)
    for (int j=0; j < 2; j++)
      M[{i, j}] = i+j;
  Vector<double> f({1, 2});
  // matvec product
  auto f2 = M.matvec(f);
  assert (f2.length == 3);
  assert (abs(f2[2] - 8) < eps);

  // conj gradient known solution (Wikipedia)
  {
  Matrix<double> M2(2,2);
  M2[{0,0}] = 4;
  M2[{0,1}] = 1;
  M2[{1,0}] = 1;
  M2[{1,1}] = 3;
  Vector<double> x({2,1});
  Vector<double> b({1,2});
  int maxiter = 5;
  double tol = 1e-08;
  assert (cg(M2, b, x, tol, maxiter) == 2);
  assert (abs(x[0] - 0.0909) < 1e-05);
  assert (abs(x[1] - 0.6364) < 1e-05);
  }

```

```
  Heat1D H(0.3125, 3, 0.1);
  std::cout << "Compare (how it should be):\n"
        << "[[ 2, -0.5,    0]\n [-0.5,  2, -0.5]\n"
        << " [   0, -0.5,  2]]\nwith:" << std::endl;
  H.show_mat();

  Heat2D H2(0.3125, 3, 0.1);
  std::cout << "Compare:" << std::endl;
  H2.show_mat();
  std::cout << "with what is given in the assignment" << std::endl;

  std::cout << "All tests passed!\n" << std::endl;
#endif

  double alpha = 0.3125;
  double dt = 0.001;
  int m = 99;
  Vector<double> u_0(u_init(m,1));
  double t_end = 1;
  std::cout << "1D case with alpha = " << alpha << ", m = " << m << ", dt = " << dt << std::
    endl;
  Heat1D H1D(alpha, m, dt);
  Vector<double> sol_ex = H1D.exact(t_end, u_0);
  Vector<double> sol_nu = H1D.solve(t_end, u_0);
  double rmse = RMS(sol_ex - sol_nu);
  std::cout << "RMSE of (exact - numerical) solution: " << rmse << std::endl;

  dt = 0.001;
  t_end = 0.5;
  u_0 = u_init(m,2);
  std::cout << "\n2D case with alpha = " << alpha << ", m = " << m << ", dt = " << dt << std
    ::endl;
  Heat2D H2D(alpha, m, dt);
  sol_ex = H2D.exact(t_end, u_0);
  sol_nu = H2D.solve(t_end, u_0);
  rmse = RMS(sol_ex - sol_nu);
  std::cout << "RMSE of (exact - numerical) solution: " << rmse << std::endl;
  return 0;
}
```

Listing A.1: C++ code used to compute the numerical solution of the Heat equation