The Conjugate Gradient Method

Will Woolfenden

November 12, 2023

1 Introduction

The Conjugate Gradient (CG) method is a method for efficiently solving a system of linear equations. It is efficient because time complexity is reduced from other methods, such as gradient descent or Newton's method. Linear systems with sparse matrices are common when we apply finite difference methods to solve boundary value problems. In this report we look at solving these linear systems, providing finite difference approximations to Poisson's equation in 1D $u''(s) = g(s)$ and in 2D $\nabla^2 u(s,t) = 1$.

2 Methods, Results and Verification

2.1 Conjugate Gradient

In CG, we start with a guess $\mathbf{x}_0$ to solve $\mathbf{Ax} = \mathbf{b}$. At every step, we compute $\mathbf{r}_i = \mathbf{Ax}_i - \mathbf{b}$ with stopping criterion $||\mathbf{r}|| < \varepsilon$. Our search vectors $\mathbf{p}_i$ are chosen such that they obey $\mathbf{p}_j^{\mathrm{T}}\mathbf{Ap}_i$ for $i \neq j$, with $\mathbf{p}_0 = \mathbf{r}_0$. We perform this with a Gram-Schmidt orthogonalisation[1] with the inner product $\langle \mathbf{u}, \mathbf{v} \rangle_{\mathbf{A}} = \mathbf{u}^{\mathrm{T}}\mathbf{Av}$. Each iteration performs

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \frac{\mathbf{r}_i^{\mathrm{T}}\mathbf{r}_i}{\mathbf{p}_i^{\mathrm{T}}\mathbf{Ap}_i}\mathbf{p}_i$$

$$\mathbf{r}_{i+1} = \mathbf{r}_i - \frac{\mathbf{r}_i^{\mathrm{T}}\mathbf{r}_i}{\mathbf{p}_i^{\mathrm{T}}\mathbf{Ap}_i}\mathbf{Ap}_i$$

$$\mathbf{p}_{i+1} = \mathbf{r}_{i+1} + \frac{\mathbf{r}_{i+1}^{\mathrm{T}}\mathbf{r}_{i+1}}{\mathbf{r}_i^{\mathrm{T}}\mathbf{r}_i}\mathbf{p}_i.$$

The residual vector $\mathbf{r}_i$ describes the distance of $\mathbf{x}_i$ to the solution $\mathbf{x}*$ which satisfies $\mathbf{Ax}* = \mathbf{b}$. The vectors $\mathbf{p}_i$ are the search directions of the iteration. For every iteration, we compute the distance we need to travel along each search direction, by intuitively realising that at a point $\mathbf{x}_i$, the vector from $\mathbf{x}_i$ to the optimiser $\mathbf{x}*$ is A-orthogonal to all the search directions we have already traversed.

2.2 Finite Difference Methods: Poisson's Equation

We want to form a finite difference approximation to Poisson's equation in 1D $u''(s) = g(s)$, subject to boundary conditions $u(0) = u(1) = 0$. We discretise s into $n + 2$ points[2] in the interval $[0, 1]$

[1] this requires $\mathbf{A}$ to be $n \times n$ with all vectors being n-vectors.

[2] n points excluding 0 and 1.

where $s_k = kh,\ k = 0, 1, 2, \ldots n+1$ and $h = 1/(n+1)$. The finite difference approximation to the second derivative is

$$u''(s) \approx \frac{u(s-h) - 2u(s) + u(s+h)}{h^2} \tag{1}$$

which can be seen by the Taylor series expansions of the $u(s)$ terms. Write $u_k = u(s_k),\ g_k = g(s_k)$. Boundary conditions are $u_0 = u_n = 0$. We obtain the linear system

$$\begin{bmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & 1 & \ddots & \ddots & & \\ & & \ddots & \ddots & 1 & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -2 \end{bmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ u_{n-1} \\ u_n \end{pmatrix} = -h^2 \begin{pmatrix} g_1 \\ g_2 \\ \vdots \\ \vdots \\ g_{n-1} \\ g_n \end{pmatrix}.$$

This is a linear system of the form $\mathbf{Ax} = \mathbf{b}$ which we can solve using the CG method. Consider the case where g is the constant function $g(s) = -1$, and where $\mathbf{A}$ is 5×5. The system in full, changing sign, is

$$\begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{pmatrix} = \frac{1}{(n+1)^2} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}.$$

Our code produces `tridiagonal_convergence_0611.txt`:

```
      Time t |      L2 norm |    LInf norm
0 |      2.22448 |      1.97222
5.4392e-05 |   0.588795 |   0.397175
6.394e-05 |   0.233913 |   0.188237
7.0762e-05 |    0.10518 |  0.0963511
7.7545e-05 | 0.00941914 | 0.00515077
8.3386e-05 | 2.94053e-16 | 1.73472e-16
```

which claims to solve the system in exactly 5 iterations. This is appropriate for CG, which is designed to converge in no more iterations than the size of the system [3]. Our program prints the solution $x = (0.0694444, 0.111111, 0.125, 0.111111, 0.0694444)$. Using the solver in `Octave` verifies this computation, shown in B.1, hence it is the true solution of the linear system.

The ODE we are solving is $u''(s) = -1$. By direct integration, the general solution is $u(s) = Cs^2 + Ds$ for constants C, D. Plugging in boundary conditions, we obtain the particular solution $u(s) = -s(s-1)/2$ in closed form. See Figure 1 for a comparison.

2.3 Perturbed Tridiagonal

We perturb the system such that the main diagonal contains $2 + \alpha$ instead of 2, for some positive real constant α. In this case, we impose $\mathbf{b}$ to be the vector with all entries $b_i = 2.5$, equivalently all $g_i = -5(n+1)^2/2$. Figure 2 illustrates the convergence of these tridiagonal systems. We notice that, as the entries on the main diagonal grow, the effectiveness of the iterations increases. The convergence of the CG method is $\mathcal{O}(m\sqrt{\kappa})$, where m is the number of non-zero entries in the matrix and κ is the condition number for any choice of norm [4]. As the entries on the diagonal grow, $\mathbf{A}$

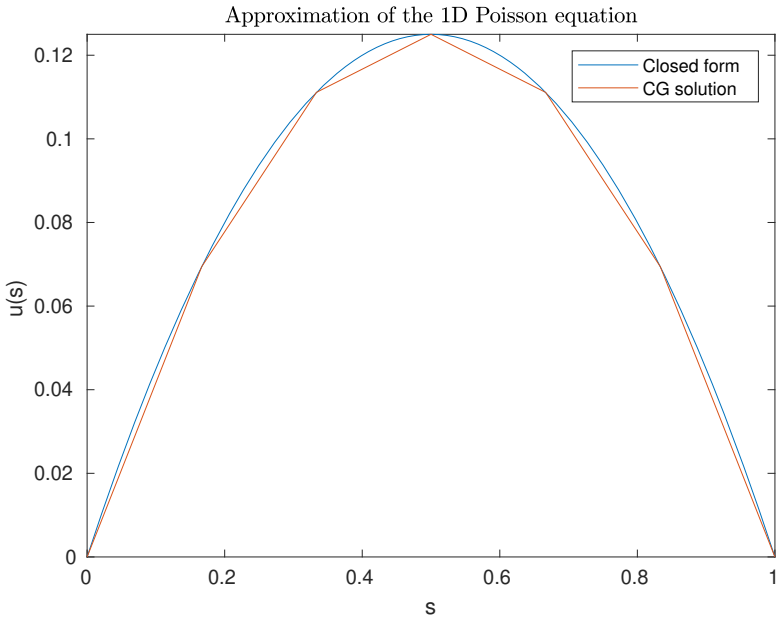

Figure 1: Comparison of the closed form solution to the ODE, using `fplot()`, and the $n = 5$ finite difference approximation from solving the system with CG.

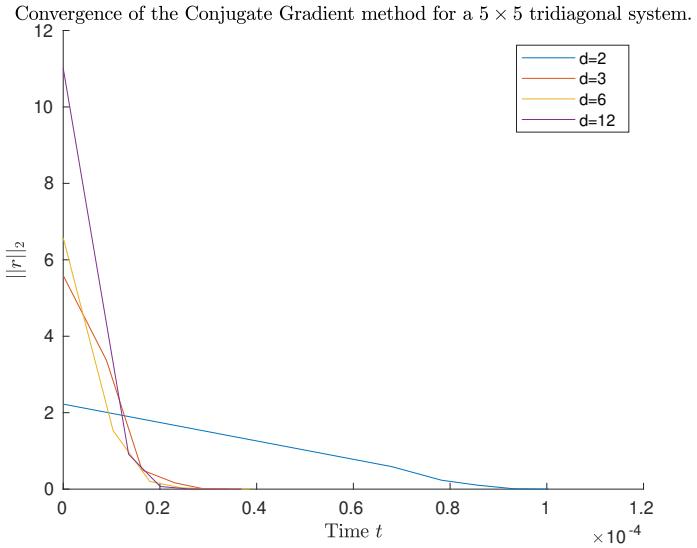

Figure 2: Convergence graphs for different size tridiagonal systems. The solutions converge more rapidly for larger diagonal entries. The matrices $\mathbf{A}$ are the same as earlier except for $2 + \alpha$ on the diagonal, and the vectors $\mathbf{b}$ are $b_k = 2.5$ for all k.

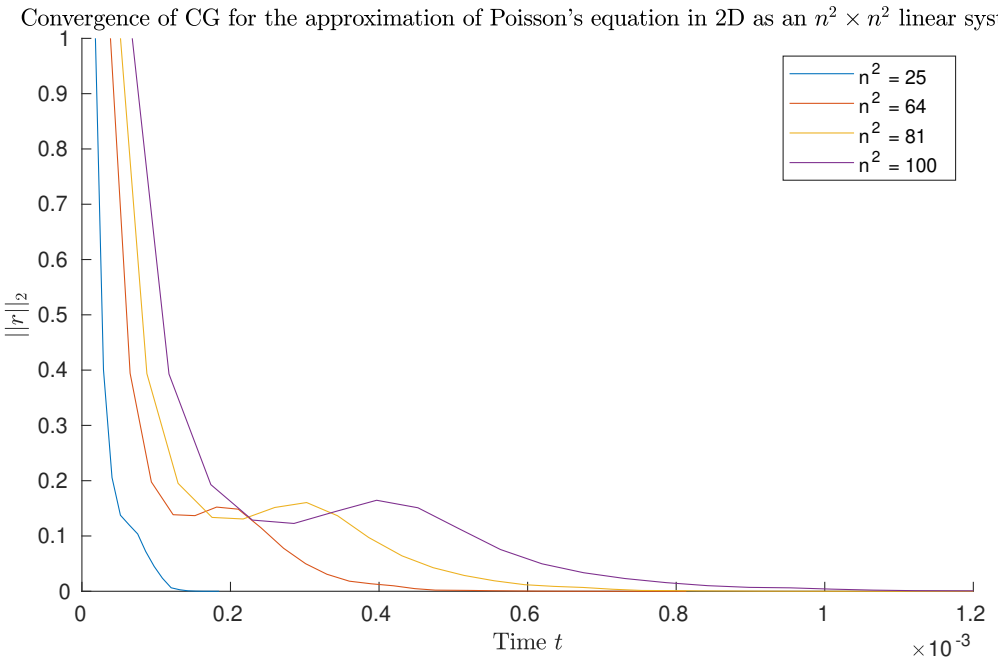

Figure 3: Convergence graphs for the CG method applied to the linear system with the Laplacian matrix. Each path attains a unique minimum and maximum which bound a region of inflection.

becomes closer to a diagonal matrix, and its condition number decreases, hence CG is more effective. Important to note is that our implementation does not entirely match this claim on the order of the CG method. Our `MMatrix` and `MVector` classes are not optimised for speed. For example, we are not optimised for multiplying with a sparse matrix, which would allow us to ignore all the entries containing 0.

2.4 Poisson's Equation in 2D

We are now interested in the problem in two variables $\nabla^2 u(s,t) = 1$, with boundary conditions $u(s,0) = u(s,1) = u(0,t) = u(1,t) = 0$. By the finite difference method, the matrix is of the form

$$(\mathbf{A})_{ij} = \begin{cases} 4 & \text{if } i = j \\ -1 & \text{if } |i-j| = n \\ -1 & \text{if } |i-j| = 1 \text{ and} \\ & (i+j) \mod (2n) \neq 2n-1 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

which we will refer to as the *Laplacian matrix*. Like before, $\mathbf{b} = b_i = 1/(n+1)^2$. See Figure 3, which shows a selection of iterations for the 2D approximation to Poisson's equation. In Figures 4 and 5 we show the impacts of scaling the matrix size. Particularly in Figure 5, we have shown that our implementation is $\mathcal{O}(d^3)$, where $d = n^2$ is the size of the system. Methods for solving a system of n linear equations are often $\mathcal{O}(n^3)$ [2]. Solutions to Poisson's equation in 2D are shown in Figure

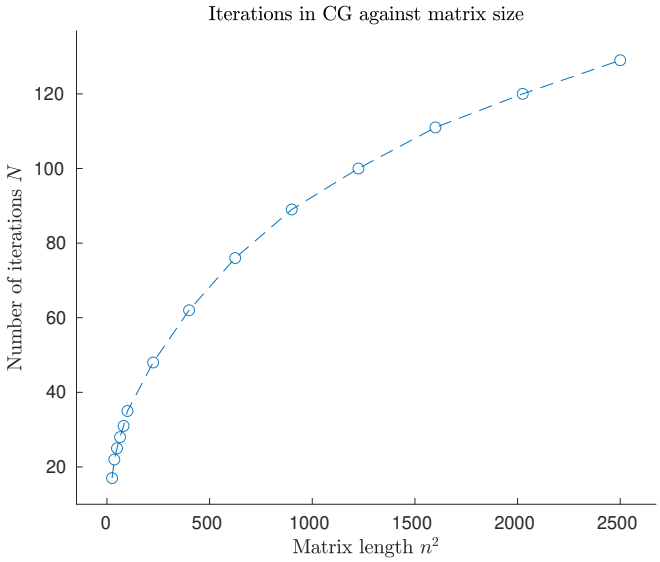

Figure 4: Number of iterations required for convergence for different matrix sizes. The curve appears logarithmic

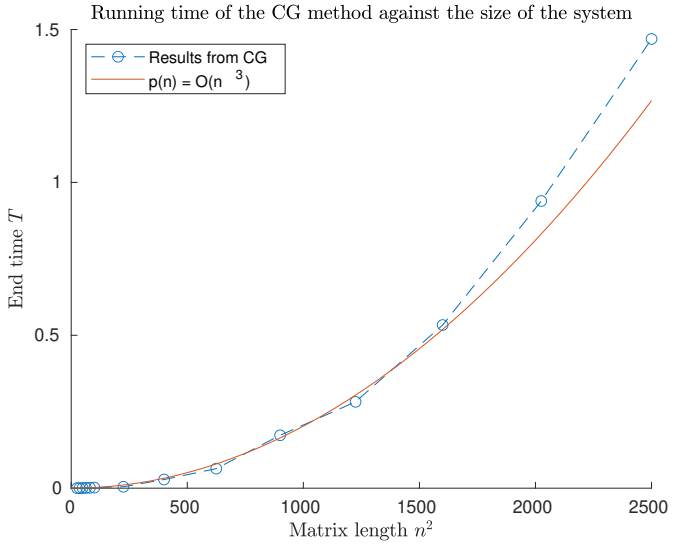

Figure 5: Running time, measured in seconds, of the C++ implementation of the CG method, against the size of the linear system we are solving. The polynomial approximation to our results was performed in MATLAB, and the fitting was unsuccessful for polynomials of degree < 3. The polynomial is technically $\mathcal{O}(n^6)$ agreeing with our definition of n such that the system is $n^2 \times n^2$, but it is a degree 3 polynomial on the domain.

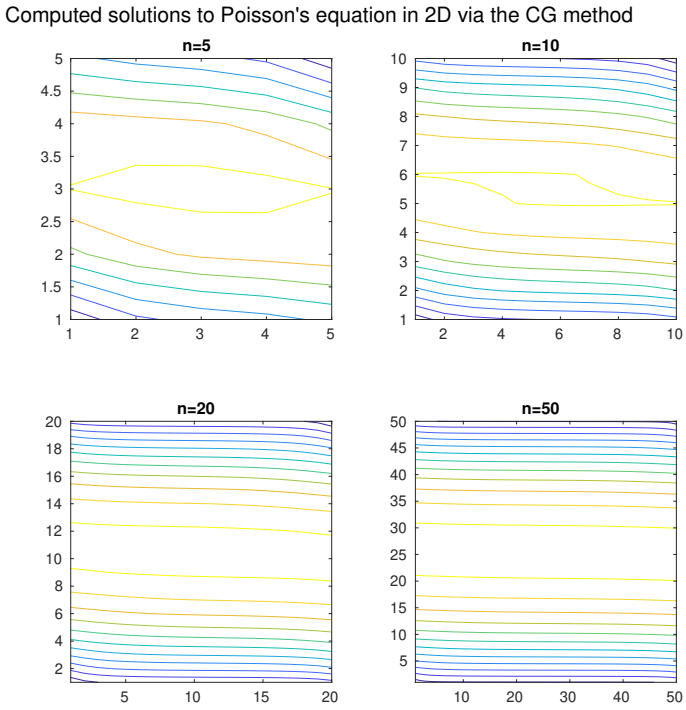

Figure 6: Solutions to Poisson's equation in 2D computed as a solution to a system of linear equations via. the CG method. As n increases, our results converge and we see negligible change between $n = 20$ and $n = 50$. Blue contours are lower in magnitude than the yellow curves near the centres of the images.

6, where we have shown a convergence of results as n increases. The boundary condition is attained on the horizontal boundaries.

2.5 Properties of the Conjugate Gradient Method

CG is only stable in exact arithmetic for symmetric, positive definite matrices [1]. As such, the modified inner product from earlier of a vector with itself $\langle \mathbf{p}, \mathbf{p} \rangle_{\mathbf{A}} = \mathbf{p}^{\mathrm{T}}\mathbf{A}\mathbf{p}$ is always positive. Furthermore if $\mathbf{A}$ is not symmetric then

$$\begin{aligned} \langle \mathbf{u}, \mathbf{v} \rangle_{\mathbf{A}} &= \langle \mathbf{u}, \mathbf{A}\mathbf{v} \rangle \\ &= \langle \mathbf{A}^{\mathrm{T}}\mathbf{u}, \mathbf{v} \rangle \\ &\neq \langle \mathbf{A}\mathbf{u}, \mathbf{v} \rangle \end{aligned}$$

is not an inner product.

All the stencil matrices we have looked at are symmetric (by inspection) and positive definite since they have entirely positive eigenvalues. This is a necessary criterion for CG to succeed. Floating point arithmetic may lead to rounding errors during computation, meaning that some methods will not succeed as they may in exact arithmetic.

Given the 5×5 matrix

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix}$$

we can compute its inverse

$$\mathbf{A}^{-1} = \begin{bmatrix} \frac{5}{6} & \frac{2}{3} & \frac{1}{2} & \frac{1}{3} & \frac{1}{6} \\ \frac{2}{3} & \frac{4}{3} & 1 & \frac{2}{3} & \frac{1}{3} \\ \frac{1}{2} & 1 & \frac{3}{2} & 1 & \frac{1}{2} \\ \frac{1}{3} & \frac{2}{3} & 1 & \frac{4}{3} & \frac{2}{3} \\ \frac{1}{6} & \frac{1}{3} & \frac{1}{2} & \frac{2}{3} & \frac{5}{6} \end{bmatrix}$$

and find the condition number $||\mathbf{A}||_F||\mathbf{A}^{-1}||_F \approx 20.7$. The identity matrix has the minimum condition number 1. As we saw earlier, the condition number affects the time of the CG method. Particularly, Figure 2 showed that for larger diagonal entries, the method converges faster. For larger diagonal entries, the condition number is smaller. Hence a large condition number can be a strong factor affecting the run time of CG.

3 Conclusion

We can use the conjugate gradient method to solve a linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ for a symmetric positive definite matrix. Uniquely, CG will always converge within the number of iterations that there are rows/columns in the matrix. It does this by computing search directions which are conjugate under $\mathbf{A}$ and traversing them, computing how far each direction must be travelled by. Conjugate Gradient may be favourable to methods such as gradient descent or other options when we can guarantee stability. The time complexity of the method is improved for a sparse and also a well-conditioned matrix. Our results show the effectiveness of the CG method for solving finite difference approximations to differential equations in one and two dimensions, where we have been able to show accuracy and convergence of our results.

References

[1] Anne Greenbaum. “Chapter 4 - Effects of Finite Precision Arithmetic”. In: *Iterative methods for solving linear systems*. SIAM, 1997.

[2] C. D. Meyer. “Linear Equations”. In: *Matrix analysis and applied linear algebra*. Society for Industrial and Applied Mathematics, 2000, pp. 14–16.

[3] Jonathan Richard Shewchuk. “An introduction to the conjugate gradient method without the agonizing pain”. In: (1994), pp. 30–31.

[4] Jonathan Richard Shewchuk. “An introduction to the conjugate gradient method without the agonizing pain”. In: (1994), pp. 37–38.

A Appendix - C++ Code Implementations

A.1 Compiled Code

A.1.1 `main.cpp`

```cpp
#include <vector>
#include <algorithm>
#include <iostream>
#include <fstream>
#include <cmath>
#include <iomanip>
#include <string>
#include <chrono>
#include <cstddef>
#include <ctime>
#include <ratio>

#include "mvector.h"
#include "mmatrix.h"

using namespace std;
using namespace chrono;

MVector conjugateGradientSolve(const MMatrix&, const MVector&, const MVector&, const string filename);
void computeLaplacian(int n, const string& filename);

int main()
{
    MVector b(5, 1.0/36.0);

    MMatrix A(5, 5);
    makeTridiagonal(A, 2, -1);

    // starting guess is elementary basis vector
    MVector x_0 = {1,0,0,0,0};

    //regular 5x5 solve (3.35-3.36)
    MVector x = conjugateGradientSolve(A, b, x_0, "tridiagonal_convergence_0611.txt");
    cout << x << endl;

    MMatrix A1(5, 5), A2(5, 5), A3(5, 5);
```

```
    makeTridiagonal(A1, 2+1, -1);
    makeTridiagonal(A2, 2+4, -1);
    makeTridiagonal(A3, 2+10, -1);

    MVector b1(5, 2.5);

    // perform CG for different values of alpha (3.37)
    x = conjugateGradientSolve(A1, b1, x_0, "tridiagonal_a1_conv.txt");
    x = conjugateGradientSolve(A2, b1, x_0, "tridiagonal_a4_conv.txt");
    x = conjugateGradientSolve(A3, b1, x_0, "tridiagonal_a10_conv.txt");

    MMatrix P(16, 16);
    makeLaplacian(P, 4, -1);
    cout << P << endl;

    // Laplace's equation (3.39)
    computeLaplacian(5, "lap_conv_n5_0611.txt");
    computeLaplacian(6, "lap_conv_n6_0611.txt");
    computeLaplacian(7, "lap_conv_n7_0611.txt");
    computeLaplacian(8, "lap_conv_n8_0611.txt");
    computeLaplacian(9, "lap_conv_n9_0611.txt");
    computeLaplacian(10, "lap_conv_n10_0611.txt");
    computeLaplacian(20, "lap_conv_n20_0611.txt");
    computeLaplacian(50, "lap_conv_n50_0611.txt");


    return 0;
}

// conjugate gradient method implementation
MVector conjugateGradientSolve(const MMatrix& A, const MVector& b, const MVector& x_0
    , const string filename = "iteration.txt")
{
    // initialisations
    int maxIterations = 1000;
    double tolerance = 1e-6;

    if (A.Rows() != A.Cols())
    {
        throw invalid_argument("error: matrix is not square");
    }
    int dim = A.Rows();

    MVector rPrev(dim), pPrev(dim), xPrev(dim), r(dim), p(dim), x(dim);
    double alpha, beta;

    // solver writes a file, an optional filename argument is given
    ofstream of;
    int w = 12;
    of.open(filename);
    if (!of)
    {
        throw invalid_argument("error: failed to open file");
    }
    of << setw(w) << "Time t" << " | "
    << setw(w) <<  "L2 norm" << " | "
    << setw(w) << "LInf norm" << "\n";

    xPrev = x_0;
    rPrev = b - A*xPrev;
```

```
    pPrev = rPrev;

    high_resolution_clock::time_point t1 = high_resolution_clock::now();

    of << setw(w) << 0 << " | "
    << setw(w) <<  rPrev.L2Norm() << " | "
    << setw(w) << rPrev.LInfNorm() << "\n";

    for (int iter=0; iter<maxIterations; iter++)
    {
        // ...calculate new values for x and r here...
        alpha = dot(rPrev, rPrev)/dot(pPrev, A*pPrev);
        x = xPrev + alpha*pPrev;
        r = rPrev - alpha*(A*pPrev);

        // write the iteration
        high_resolution_clock::time_point t2 = high_resolution_clock::now();
        duration<double> time_span = duration_cast<duration<double>>(t2 - t1);

        of << setw(10) << time_span.count() << " | "
        << setw(10) <<  r.L2Norm() << " | "
        << setw(10) << r.LInfNorm() << "\n";

        // check if solution is accurate enough
        if (r.L2Norm() < tolerance) break;

        // ...calculate new conjugate vector p here...
        beta = dot(r, r)/dot(rPrev, rPrev);
        p = r + beta*pPrev;

        // step the placeholder variables for the next iteration
        pPrev = p;
        rPrev = r;
        xPrev = x;
    }

    of.close();
    return x;
}

// procedure to compute a solution using the conjugate gradient method, given the
    value n and the file to write to
void computeLaplacian(int n, const string& filename)
{
    MMatrix H(n*n, n*n);
    makeLaplacian(H, 4, -1);

    MVector a(n*n, 1.0/pow(n+1, 2));
    // initial guess is always the unit vector
    MVector x(n*n);
    x[0] = 1;

    conjugateGradientSolve(H, a, x, filename);
}
```

A.1.2 mmatrix.cpp

```
#include <vector>
#include <iostream>
#include <iomanip>
#include <ios>
```

```cpp
#include <algorithm>
#include <cmath>

#include "mmatrix.h"
#include "mvector.h"

using namespace std;

MVector operator*(const MMatrix& A, const MVector& v)
{
  // check compatibility
  if (A.Cols() != v.size())
  {
    cout << "wrong dims" << endl;
    exception e;
    throw e;
  }
  // initialise output vector
  int vout_size = A.Rows();
  MVector vout = MVector(vout_size);

  for (int i = 0; i < A.Rows(); i++)
  {
    // sum across j
    double sum = 0;
    for (int j = 0; j < A.Cols(); j++)
    {
      sum += A(i, j)*v[j];
    }
    vout[i] = sum;
  }
  return vout;
}

ostream& operator<<(ostream& os, const MMatrix& A)
{
  for (int i = 0; i < A.Rows(); i++)
  {
    os << "|  ";
    for (int j = 0; j < A.Cols(); j++)
    {
      os << setw(3) << A(i, j) << "  ";
    }
    os << "|\n";
  }
  return os;
}

void makeTridiagonal(MMatrix& A, double d, double bd)
{
  /*
  takes the arguments for the diagonal and band, and passes the matrix by reference.
  A must already be the size we want, but its entries will be entirely overwritten.
  */
  if (A.Rows() != A.Cols())
  {
    throw invalid_argument("error: matrix not square");
  }
  for (int i = 0; i < A.Rows(); i++)
  {
```

```
    for (int j = 0; j < A.Cols(); j++)
    {
      if (i == j)
      {
        A(i, j) = d;
      }
      else if (abs(i - j) == 1)
      {
        A(i, j) = bd;
      }
      else
      {
        A(i, j) = 0;
      }
    }
  }
}

void makeLaplacian(MMatrix& A, double d, double bd)
{
  /*
  behaves similarly to makeTridiagonal in its arguments. for solutions to the
    Laplacian equation.
  */
  if (A.Rows() != A.Cols())
  {
    throw invalid_argument("error: matrix not square");
  }
  double sz = sqrt(A.Rows());
  if (floor(sz) != sz)
  {
    // A must be n^2 by n^2
    throw invalid_argument("error: matrix is not of dimension n^2.n^2");
  }
  int n = sz;
  for (int i = 0; i < A.Rows(); i++)
  {
    for (int j = 0; j < A.Cols(); j++)
    {
      if (i == j)
      {
        A(i, j) = d;
      }
      else if (abs(i - j) == n)
      {
        A(i, j) = bd;
      }
      else if ((abs(i - j) == 1) && ((i + j) % 2*n != 2*n - 1))
      {
        A(i, j) = bd;
      }
      else
      {
        A(i, j) = 0;
      }
    }
  }
}
```

A.1.3 mvector.cpp

```
#include <vector>
#include <iostream>
#include <iomanip>
#include <algorithm>
#include <cmath>

#include "mvector.h"

using namespace std;

// constructors in header

MVector operator*(const double& lhs, const MVector& rhs)
{
  MVector temp = rhs;
  for (int i = 0; i < temp.size(); i++)
  {
    temp[i] *= lhs;
  }
  return temp;
}

MVector operator*(const MVector& lhs, const double& rhs)
{
  return rhs*lhs;
}

MVector operator+(const MVector& lhs, const MVector& rhs)
{
  if (lhs.size() != rhs.size())
  {
    exception e;
    throw e;
  }
  MVector temp(lhs.size());
  for (int i = 0; i < temp.size(); i++)
  {
    temp[i] = lhs[i] + rhs[i];
  }
  return temp;
}

MVector operator-(const MVector& lhs, const MVector& rhs)
{
  return lhs+(-1*rhs);
}

// alternative overload such that given MVector v, -v is allowed
MVector MVector::operator-()
{
  MVector vec(v.size());
  for (int i = 0; i < vec.size(); i++)
  {
    vec[i] = -v[i];
  }
  return vec;
}

```

```
MVector operator/(const MVector& lhs, const double& rhs)
{
  return (1/rhs)*lhs;
}

ostream& operator<<(ostream& os, const MVector& v)
{
  int n = v.size();
  os << "(";
  for (int i = 0; i < n-1; i++)
  {
    os << setw(10) << v[i] << ", ";
  }
  os << v[n-1];
  os << ")";
  return os;
}

double MVector::LInfNorm() const
{
  double maxAbs = 0;
  size_t s = size();
  for (int i=0; i<s; i++)
  {
    maxAbs = max(abs(v[i]), maxAbs);
  }
  return maxAbs;
}

double MVector::L2Norm() const
{
  double sum = 0;
  for (int i = 0; i < v.size(); i++)
  {
    sum += v[i]*v[i];
  }
  return sqrt(sum);
}

double dot(const MVector& lhs, const MVector& rhs)
{
  if (lhs.size() != rhs.size())
  {
    throw invalid_argument("error: dot product not defined for vectors of non-equal
    dimension");
  }
  double sum = 0;
  for (int i = 0; i < lhs.size(); i++)
  {
    sum += lhs[i]*rhs[i];
  }
  return sum;
}
```

A.2 Included Code

A.2.1 mmatrix.h

```
ifndef MMATRIX_H // the 'include guard'
```

```
#define MMATRIX_H

#include <vector>
#include <iostream>
#include "mvector.h"

using namespace std;

// Class that represents a mathematical matrix
class MMatrix
{
public:
  // constructors
  MMatrix() : nRows(0), nCols(0) {}
  MMatrix(int n, int m, double x = 0) : nRows(n), nCols(m), A(n * m, x) {}

  // set all matrix entries equal to a double
  MMatrix &operator=(double x)
  {
    for (unsigned i = 0; i < nRows * nCols; i++) A[i] = x;
    return *this;
  }

  // access element, indexed by (row, column) [rvalue]
  double operator()(int i, int j) const
  {
    return A[j + i * nCols];
  }

  // access element, indexed by (row, column) [lvalue]
  double &operator()(int i, int j)
  {
    return A[j + i * nCols];
  }

  friend MVector operator*(const MMatrix& A, const MVector& v);
  friend ostream& operator<<(ostream& os, const MMatrix& A);

  // size of matrix
  int Rows() const { return nRows; }
  int Cols() const { return nCols; }

private:
  unsigned int nRows, nCols;
  vector<double> A;
};

void makeTridiagonal(MMatrix& A, double d, double bd);
void makeLaplacian(MMatrix& A, double d, double bd);

#endif
```

A.2.2 mvector.h

```
#ifndef MVECTOR_H // the 'include guard'
#define MVECTOR_H // see C++ Primer Sec. 2.9.2

#include <vector>

```

```
using namespace std;

// Class that represents a mathematical vector
class MVector
{
public:
  // constructors
  MVector() {}
  MVector(int n) : v(n) {}
  MVector(int n, double x) : v(n, x) {}
  MVector(initializer_list<double> l) : v(l) {}

  // access element (lvalue) (see example sheet 5, q5.6)
  double &operator[](int index)
  {
    return v[index];
  }

  // access element (rvalue) (see example sheet 5, q5.7)
  double operator[](int index) const {
    return v[index];
  }

  // operator overloads
  MVector operator-();

  friend MVector operator*(const double& lhs, const MVector& rhs);
  friend MVector operator*(const MVector& lhs, const double& rhs);
  friend MVector operator+(const MVector& lhs, const MVector& rhs);
  friend MVector operator-(const MVector& lhs, const MVector& rhs);
  friend MVector operator/(const MVector& lhs, const double& rhs);

  friend ostream& operator<<(ostream& os, const MVector& v);

  // get size
  int size() const { return v.size(); } // number of elements

  //vector norms
  double LInfNorm() const;
  double L2Norm() const;


private:
  vector<double> v;
};

// dot product
double dot(const MVector& lhs, const MVector& rhs);
```

B Appendix - Extra Code

B.1 Octave verification of linear solver

```
octave:11> A
A =
2  -1   0   0   0
-1   2  -1   0   0
```

```
0  -1   2  -1   0
0   0  -1   2  -1
0   0   0  -1   2

octave:12> b
b =
0.027778
0.027778
0.027778
0.027778
0.027778

octave:13> A\b
ans =
0.069444
0.111111
0.125000
0.111111
0.069444
```

C Appendix - Notes on the problems addressed

The code provided does not immediately provide all of the results stated. I have enclosed the code as is due to time constraints. The current implementation writes a file which documents computation time for any execution of `conjugateGradientSolve()`. It is stated in the body that our code produces a file which consists of given results. The norm values are consistent, but the times will change due to different hardware and system resources if executed. The code was modified when it was needed to write the solutions to Poisson's equation in 2D in matrix form.

There is confusion with the problem for Poisson's equation in 2D. The problem states that we are solving $\nabla^2 u = 1$ and the formulation for this is analogous to the 1D problem given, however for the 1D problem the model equation is $u'(s) = -1$. Our solutions for both are positive valued. If we are solving the 2D equation our solution should be negative, but the matrix and vector in the solution are verified to be correct as given by the problem statement. Furthermore, the solution seems to only satisfy the boundary conditions on the horizontal boundary lines.

All code compiled successfully on Ubuntu 22.04 Linux with g++ using `-O2` optimisation, faster floating point arithmetic `-ffast-math` and native hardware priority `-march=native`.