7.2 Boundary-value problems

In this topic, we will describe and look at approximating solutions to ordinary differential equations using the shooting method, followed by observing how the shooting method can be very efficient if the ODE is linear. We will then look at an alternative approach to approximating solutions to LODEs by using the finite-difference method. From this, we will observe that this requires solving systems of linear equations, so we will look at how we can efficient solve a tri-diagonal system of linear equations, but we will then also look at a more general sparse matrix representation.

The shooting method for ODEs

A boundary-value problem is a problem where the value of the function is known or prescribed at two points, and the behaviour of the solution is described by a 2nd-order ordinary differential equation between those two points:

$$\begin{aligned} u^{(2)}(x) &= f\left(x, u(x), u^{(1)}(x)\right) \\ u(a) &= u_a \\ u(b) &= u_b \end{aligned}$$

In this case, we can use the *shooting method* to approximate a solution to this boundary-value problem. For this technique, we convert the boundary-value problem into an initial-value problem:

$$\begin{aligned} u_0^{(2)}(x) &= f\left(x, u_0(x), u_0^{(1)}(x)\right) \\ u_0(a) &= u_a \\ u_0^{(1)}(a) &= \frac{u_b - u_a}{b-a} = s_0 \end{aligned}$$

The initial slope is approximated by the slope between the two end points. You now find a solution to this using, for example, the secant method:

1. Find a solution $u_0(x)$ to the initial-value problem and find $u_0(b)$. Compare this with u_b and adjust your slope appropriate to create a new initial-value problem
$$\begin{aligned} u_1^{(2)}(x) &= f\left(x, u_0(x), u_0^{(1)}(x)\right) \\ u_1(a) &= u_a \\ u_1^{(1)}(a) &= \frac{\left(2u_b - u_0(b)\right) - u_a}{b-a} = s_1 \end{aligned}$$
where the initial slope is now corrected.
2. Find a solution $u_1(x)$ to the initial-value problem and find $u_1(b)$.

3. We now define the function $\hat{u}_b(s)$ to be the value at $x = b$ of the solution to the IVP with the initial slope s. For the two approximations we have already calculated, $\hat{u}_b(s_0) = u_0(b)$ and $\hat{u}_b(s_1) = u_1(b)$. We want to find that s^* such that $\hat{u}_b(s^*) = u_b$, or in other words, $\hat{u}_b(s^*) - u_b = 0$. This is a root-finding problem, and we now have two approximations, s_0 and s_1 to that root, so we may proceed from here by using the secant method. Thus, given s_{k-1} and s_k, the next slope we try is

$$s_{k+1} = \frac{s_{k-1}\left(\hat{u}_b(s_k) - u_b\right) - s_k\left(\hat{u}_b(s_{k-1}) - u_b\right)}{\left(\hat{u}_b(s_k) - u_b\right) - \left(\hat{u}_b(s_{k-1}) - u_b\right)}$$
$$= \frac{s_{k-1}\left(\hat{u}_b(s_k) - u_b\right) - s_k\left(\hat{u}_b(s_{k-1}) - u_b\right)}{\hat{u}_b(s_k) - \hat{u}_b(s_{k-1})}$$

We solve this initial-value problem and calculate $\hat{u}_b(s_{k+1}) - u_b$ and determine whether we must continue iterating.

Problems

1. Write the system of first-order linear equations necessary to use the shooting method for the BVP

$$u^{(2)}(x) = xu(x)u^{(1)}(x) + 1$$
$$u(2.3) = 4.79$$
$$u(5.6) = 1.08$$

The shooting method for linear ODEs

There is one special case where we need not iterate: when the 2nd-order ordinary differential equation is linear:

$$u^{(2)}(x) + a_1(x)u^{(1)}(x) + a_0(x)u(x) = g(x).$$

In this case, it is guaranteed that the IVP with the initial slope $u^{(1)}(a) = s_2$ will give the same solution as the BVP.

Problems

1. Write the system of first-order IVPs necessary to use the shooting method for the BVP

$$u^{(2)}(x) - 3u^{(1)}(x) + 2u(x) = 1$$
$$u(2.3) = 4.79$$
$$u(5.6) = 1.08$$

Finite-difference method

Consider the following 2nd-order LODE:

$$u^{(2)}(x)+a_1(x)u^{(1)}(x)+a_0(x)u(x)=g(x).$$

We have 2nd-order approximations of both the derivative and second derivative:

$$u^{(1)}(x)=\frac{u(x+h)-u(x-h)}{2h}+O(h^2)$$
$$u^{(2)}(x)=\frac{u(x+h)-2u(x)+u(x-h)}{h^2}+O(h^2)$$

We could substitute these two finite-difference approximations into our differential equation:

$$\frac{u(x+h)-2u(x)+u(x-h)}{h^2}+a_1(x)\frac{u(x+h)-u(x-h)}{2h}+a_0(x)u(x)\approx g(x)$$

Note that we can collect similar terms:

$$\frac{u(x+h)}{h^2}-\frac{2u(x)}{h^2}+\frac{u(x-h)}{h^2}+a_1(x)\frac{u(x+h)}{2h}-a_1(x)\frac{u(x-h)}{2h}+a_0(x)u(x)\approx g(x)$$
$$u(x-h)\left(\frac{1}{h^2}-\frac{a_1(x)}{2h}\right)+u(x)\left(a_0(x)-\frac{2}{h^2}\right)+u(x+h)\left(\frac{1}{h^2}+\frac{a_1(x)}{2h}\right)\approx g(x)$$

Suppose now divide the interval [*a*, *b*] into *n* points: $h=\frac{b-a}{n}$ so $x_k=a+kh$ where $x_0=a$ and $x_n=b$. In this case, we have that $u(x_k-h)=u(x_{k-1})$ and $u(x_k+h)=u(x_{k+1})$. We know that $u(x_0)=u_a$ and $u(x_n)=u_b$, but we do not have values for any of the interior points $x_1, x_2, \ldots x_{n-1}$. If we were to substitute $x = x_k$, we have

$$u_{k-1}\left(\frac{1}{h^2}-\frac{a_1(x_k)}{2h}\right)+u_k\left(a_0(x_k)-\frac{2}{h^2}\right)+u_{k+1}\left(\frac{1}{h^2}+\frac{a_1(x_k)}{2h}\right)\approx g(x_k)$$

which is a linear equation in three unknowns. Doing this for $k = 1, \ldots, n-1$, we have

$$u(x_0)\left(\frac{1}{h^2}-\frac{a_1(x_1)}{2h}\right)+u_1\left(a_0(x_1)-\frac{2}{h^2}\right)+u_2\left(\frac{1}{h^2}+\frac{a_1(x_1)}{2h}\right)\approx g(x_1)$$
$$u_1\left(\frac{1}{h^2}-\frac{a_1(x_2)}{2h}\right)+u_2\left(a_0(x_2)-\frac{2}{h^2}\right)+u_3\left(\frac{1}{h^2}+\frac{a_1(x_2)}{2h}\right)\approx g(x_2)$$
$$\vdots$$
$$u_{n-3}\left(\frac{1}{h^2}-\frac{a_1(x_{n-2})}{2h}\right)+u_{n-2}\left(a_0(x_{n-2})-\frac{2}{h^2}\right)+u_{n-1}\left(\frac{1}{h^2}+\frac{a_1(x_{n-2})}{2h}\right)\approx g(x_{n-2})$$
$$u_{n-2}\left(\frac{1}{h^2}-\frac{a_1(x_{n-1})}{2h}\right)+u_{n-1}\left(a_0(x_{n-1})-\frac{2}{h^2}\right)+u(x_n)\left(\frac{1}{h^2}+\frac{a_1(x_{n-1})}{2h}\right)\approx g(x_{n-1})$$

This gives a system of $n-1$ equations in the $n-1$ unknowns $u_1, u_2, \ldots, u_{n-2}, u_{n-1}$:

$$u_1\left(a_0(x_1)-\frac{2}{h^2}\right)+u_2\left(\frac{1}{h^2}+\frac{a_1(x_1)}{2h}\right)\approx g(x_1)-u_a\left(\frac{1}{h^2}-\frac{a_1(x_1)}{2h}\right)$$

$$u_1\left(\frac{1}{h^2}-\frac{a_1(x_2)}{2h}\right)+u_2\left(a_0(x_2)-\frac{2}{h^2}\right)+u_3\left(\frac{1}{h^2}+\frac{a_1(x_2)}{2h}\right)\approx g(x_2)$$

$$\vdots$$

$$u_{n-3}\left(\frac{1}{h^2}-\frac{a_1(x_{n-2})}{2h}\right)+u_{n-2}\left(a_0(x_{n-2})-\frac{2}{h^2}\right)+u_{n-1}\left(\frac{1}{h^2}+\frac{a_1(x_{n-2})}{2h}\right)\approx g(x_{n-2})$$

$$u_{n-2}\left(\frac{1}{h^2}-\frac{a_1(x_{n-1})}{2h}\right)+u_{n-1}\left(a_0(x_{n-1})-\frac{2}{h^2}\right)\approx g(x_{n-1})-u_b\left(\frac{1}{h^2}+\frac{a_1(x_{n-1})}{2h}\right)$$

We can write this as the augmented matrix:

$$\left(\begin{array}{ccccccc|c}
a_0(x_1)-\frac{2}{h^2} & \frac{1}{h^2}+\frac{a_1(x_1)}{2h} & 0 & \cdots & 0 & 0 & 0 & g(x_1)-u_a\left(\frac{1}{h^2}-\frac{a_1(x_1)}{2h}\right) \\
\frac{1}{h^2}-\frac{a_1(x_2)}{2h} & a_0(x_2)-\frac{2}{h^2} & \frac{1}{h^2}+\frac{a_1(x_2)}{2h} & \cdots & 0 & 0 & 0 & g(x_2) \\
0 & \frac{1}{h^2}-\frac{a_1(x_3)}{2h} & a_0(x_3)-\frac{2}{h^2} & \ddots & 0 & 0 & 0 & g(x_3) \\
\vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \ddots & a_0(x_{n-3})-\frac{2}{h^2} & \frac{1}{h^2}+\frac{a_1(x_{n-3})}{2h} & 0 & g(x_{n-3}) \\
0 & 0 & 0 & \cdots & \frac{1}{h^2}-\frac{a_1(x_{n-2})}{2h} & a_0(x_{n-2})-\frac{2}{h^2} & \frac{1}{h^2}+\frac{a_1(x_{n-2})}{2h} & g(x_{n-2}) \\
0 & 0 & 0 & \cdots & 0 & \frac{1}{h^2}-\frac{a_1(x_{n-1})}{2h} & a_0(x_{n-1})-\frac{2}{h^2} & g(x_{n-1})-u_b\left(\frac{1}{h^2}+\frac{a_1(x_{n-1})}{2h}\right)
\end{array}\right)$$

Solving this system of linear equations yields the vector

$$\mathbf{u}=\begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{n-3} \\ u_{n-2} \\ u_{n-1} \end{pmatrix}$$

Thus, the approximation of the solution to the BVP are the points $(u_a, u_1, u_2, u_3, \ldots, u_{n-3}, u_{n-2}, u_{n-1}, u_b)$.

Normally, solving a system of linear equations would require us to use a linear-algebra package; however, this is a tri-diagonal matrix, and thus the problem is greatly simplified. Consider the following:

$$\left(\begin{array}{cccccc|c} a_{1,1} & a_{1,2} & & & & & b_1 \\ a_{2,1} & a_{2,2} & a_{2,3} & & & & b_2 \\ & a_{3,2} & a_{3,3} & a_{3,4} & & & b_3 \\ & & a_{4,3} & a_{4,4} & a_{4,5} & & b_4 \\ & & & a_{5,4} & a_{5,5} & a_{5,6} & b_5 \\ & & & & a_{6,5} & a_{6,6} & b_6 \end{array}\right)$$

We must add $-\frac{a_{2,1}}{a_{1,1}}$ times Row 1 onto Row 2, resulting in

$$\left(\begin{array}{cccccc|c} a_{1,1} & a_{1,2} & & & & & b_1 \\ 0 & \tilde{a}_{2,2} & a_{2,3} & & & & \tilde{b}_2 \\ & a_{3,2} & a_{3,3} & a_{3,4} & & & b_3 \\ & & a_{4,3} & a_{4,4} & a_{4,5} & & b_4 \\ & & & a_{5,4} & a_{5,5} & a_{5,6} & b_5 \\ & & & & a_{6,5} & a_{6,6} & b_6 \end{array}\right).$$

where $\tilde{a}_{2,2}$ and $\tilde{b}_2$ denote the modified entries as a result of the row operation.

Next, we must add $-\frac{a_{3,2}}{\tilde{a}_{2,2}}$ times Row 2 onto Row 3, resulting in

$$\left(\begin{array}{cccccc|c} a_{1,1} & a_{1,2} & & & & & b_1 \\ 0 & \tilde{a}_{2,2} & a_{2,3} & & & & \tilde{b}_2 \\ & 0 & \tilde{a}_{3,3} & a_{3,4} & & & \tilde{b}_3 \\ & & a_{4,3} & a_{4,4} & a_{4,5} & & b_4 \\ & & & a_{5,4} & a_{5,5} & a_{5,6} & b_5 \\ & & & & a_{6,5} & a_{6,6} & b_6 \end{array}\right)$$

where the tildes represent modified entries. After three more steps, we have the matrix

$$\left(\begin{array}{cccccc|c} a_{1,1} & a_{1,2} & & & & & b_1 \\ 0 & \tilde{a}_{2,2} & a_{2,3} & & & & \tilde{b}_2 \\ & 0 & \tilde{a}_{3,3} & a_{3,4} & & & \tilde{b}_3 \\ & & 0 & \tilde{a}_{4,4} & a_{4,5} & & \tilde{b}_4 \\ & & & 0 & \tilde{a}_{5,5} & a_{5,6} & \tilde{b}_5 \\ & & & & 0 & \tilde{a}_{6,6} & \tilde{b}_6 \end{array}\right)$$

Now we may solve for the solution vector using backward substitution:

$$u_6 = \frac{\tilde{b}_6}{\tilde{a}_{6,6}}$$

$$u_5 = \frac{\tilde{b}_5 - a_{5,6}u_6}{\tilde{a}_{5,5}}$$

$$u_4 = \frac{\tilde{b}_4 - a_{4,5}u_5}{\tilde{a}_{4,4}}$$

$$u_3 = \frac{\tilde{b}_3 - a_{3,4}u_4}{\tilde{a}_{3,3}}$$

$$u_2 = \frac{\tilde{b}_2 - a_{2,3}u_3}{\tilde{a}_{2,2}}$$

$$u_1 = \frac{b_1 - a_{1,2}u_2}{a_{1,1}}$$

Note that the off-diagonal entries are never modified, and so they can simply be calculated when they are necessary. Thus, they need not even be stored; they need simply be calculated when needed. Thus, we proceed as follows:

1. Create two arrays of size $n - 1$ and store the diagonal entries as well as the entries of the target.

```
double h{b - a)/n;

std::vector<double> x(n + 1);

for ( std::size_t k{0}; k < n + 1; ++k ) {
    x[k] = a + k*h;
}

std::vector<double> diagonal(n - 1);
std::vector<double> target(n - 1);

for ( std::size_t k{0}; k < n - 1; ++k ) {
    diagonal[k] = a0(x[k + 1]) – 2.0/(h*h);
    target[k] = g(x[k + 1]);
}

target[0] -= u_a*(1.0/(h*h) -   a1(x[1])  /(2.0*h));
target[n - 2] -= u_b*(1.0/(h*h) + a1(x[n - 1])/(2.0*h));
```

2. Next, add the appropriate multiple of the super-diagonal entry onto the corresponding diagonal entry and the corresponding entry in the target vector.

```
for ( std::size_t k{1}; k < n + 1; ++k ) {
    double scalar{-(1.0/(h*h) - a1(x[k + 1])/(2.0*h)};

    diagonal[k] += scalar*(1.0/(h*h) + a1(x[k]));
    target[k] += scalar*target[k - 1];
}
```

3. Finally, create a target vector u of capacity $n + 1$, setting the first and last entries to u_a and u_b:

```
std::vector<double> u(n + 1);

u[0] = u_a;
u[n] = u_b;

u[n - 1] = target[n - 2]/diagonal[n - 2];

for ( std::size_t k{n – 2}; n > 0; --k ) {
    u[k] = (
        target[n - 1] - (1.0/(h*h) + a1(x[k + 1])/(2.0*h))*u[k + 1]
    )/diagonal[n - 1];
}
```

Note that the run-time of this is $\Theta(n)$, as it is only a series of for loops. Recall that in general, the operation of Gaussian elimination is $\Theta(n^3)$ while backward substitution is $\Theta(n^2)$. We are fortunate here, as this matrix not only is *sparse* (meaning, most of the off-diagonal entries are zero) but also has the special form of a tri-diagonal matrix.

Example of the finite difference method

Suppose we have a BVP similar to our previous version:

$$u^{(2)}(x) + 3u^{(1)}(x) + u(x) = \sin(\pi x)$$
$$u(0) = 1.3$$
$$u(1) = 2.7$$

Let us divide the interval [0, 1] into $n = 10$ sub-intervals. In this case, $a_0 : x \mapsto 1$, $a_1 : x \mapsto 3$ and $g : x \mapsto \sin(\pi x)$.

The width of each sub-interval is $h = \frac{1-0}{10} = 0.1$ so the points are $x_k = 0 + 0.1k$.

Problems

1. How does the system of linear equations differ for the following homogeneous and non-homogenous BVPs?

$$u^{(2)}(x) - 3u^{(1)}(x) + 2u(x) = 0 \qquad u^{(2)}(x) - 3u^{(1)}(x) + 2u(x) = 1$$
$$u(0) = 4.1 \text{ and } u(0) = 4.1$$
$$u(1) = 3.2 \qquad u(1) = 3.2$$

2. Write the system of linear equations necessary to use the finite-difference method for the BVP

$$u^{(2)}(x) - 3u^{(1)}(x) + 2u(x) = 1$$
$$u(2.3) = 4.79$$
$$u(5.6) = 1.08$$

Sparse matrices

The memory required to store an n x n matrix is $\Theta(n^2)$; however, most matrices in engineering are *sparse*, meaning that there are often only $\Theta(n)$ non-zero entries. Consequently, it would be convenient to store those non-zero entries using only $\Theta(n)$ memory. Additionally, matrix-vector multiplication involves multiplying each entry of the matrix by one entry of the vector and then summing appropriately; however, $0u_k = 0$, so multiplying by zero does nothing to the matrix-vector product. Consequently, we should also be able to reduce matrix-vector multiplication with sparse matrices down to $\Theta(n)$ time.

```
enum multiplication_t {
    FULL,
    DIAGONAL,
    OFF_DIAGONAL
};

template <std:size_t N>
class matrix {
    public:
        matrix( unsigned int cap );
        vec<N> multiply( vec<N> v, multiplication_t type );
        matrix &operator *=( double s );

    private:
        double        a_diagonals[N];
        unsigned int  a_i[N + 1];
        unsigned int  off_capacity;
        unsigned int *a_j;
        double       *a_entries;
};

template <unsigned int N>
void matrix<N>::resize( unsigned int new_cap ) {
    if ( (new_cap < a_i[N]) || (new_cap == off_capacity) ) {
        return;
    }

    a_new_j = new unsigned int[new_cap];
    a_new_entries = new double[new_cap];

    for ( unsigned int k{0}; k < a_i[N]; ++k ) {
        a_new_j[k] = a_j[k];
        a_new_entries[k] = a_entries[k];
    }

    delete[] a_j;
    delete[] a_entries;

    a_j = a_new_j;
    a_entries = a_new_entries;
    off_capacity = new_cap;
}
```

```
// Constructor
template <unsigned int N>
matrix<N>::matrix( unsigned int cap ):
off_capacity{ cap },
a_j{ new unsigned int[off_capacity] },
a_entries{ new double[off_capacity] } {
    for ( unsigned int i{0}; i < N; ++i ) {
        a_diagonals[i] = 0.0;
        a_i[i] = 0;
    }

    a_i[N] = 0;
}

// Copy constructor
template <unsigned int N>
matrix<N>::matrix( matrix<N> const &A ):
off_capacity{ A.off_capacity },
a_j{ new unsigned int[off_capacity] },
a_entries{ new double[off_capacity] } {
    for ( unsigned int i{0}; i < N; ++i ) {
        a_diagonals[i] = A.a_diagonals[i];
        a_i[i] = A.a_i[i];
    }

    a_i[N] = A.a_i[N];

    for ( unsigned int j{0}; j < a_i[N]; ++j ) {
        a_j[j] = A.a_j[j];
        a_entries[j] = A.a_entries[j];
    }
}

// Move constructor
template <unsigned int N>
matrix<N>::matrix( matrix<N> const &&A ):
off_capacity{ A.off_capacity },
a_j{ A.a_j },
a_entries{ A.a_entries } {
    for ( unsigned int i{0}; i < N; ++i ) {
        a_diagonals[i] = A.a_diagonals[i];
        a_i[i] = A.a_i[i];
    }

    A.a_j = A.a_entries = nullptr;
}

// Destructor
template <unsigned int N>
matrix<N>::~matrix() {
   delete[] a_j;
   delete[] a_entries;
}
```

Accessing an entry on the diagonal is straight-forward: return the corresponding entry in the diagonal array.

In order to access an off-diagonal entry, the entry $a_{i,j}$ must lie in the entries from `a_i[i]` up to `a_i[i + 1] - 1`. We will do a linear search through these entries determining whether or not any of the columns match the desired column. If so, return the corresponding entry; otherwise, return 0.

Note that we could do a binary search; however, the arithmetic operations in a binary search would far in exceed the effort required to check approximately five entries.

```
template <unsigned int N>
double matrix::get( int i, int j ) {
    if ( i == j ) {
        return a_diagonals[i];
    }

    for ( unsigned int k{a_i[i]}; k < a_i[i + 1]; ++k ) {
        if ( a_j[k] == j ) {
            return a_entries[k];
        } else if ( a_j[k] > j ) {
            return 0.0;
        }
    }

    return 0.0;
}
```

Setting an entry on the diagonal is straight-forward: assign the corresponding entry in the diagonal array.

If we are assigning an entry to be 0, we must remove it from the array of off-diagonal entries if it exists. If we are assigning an entry to be non-zero, if it is already assigned, we need only update the entry; however, if it has not already been set, we must insert the entry into the arrays of off-diagonal entries.

```
template <unsigned int N>
void matrix::set( int i, int j, double s ) {
    if ( i == j ) {
        a_diagonals[i] = s;
        return;
    }

    if ( s == 0.0 ) {
        for ( unsigned int k{a_i[i]}; k < a_i[i + 1]; ++k ) {
            if ( a_j[k] == j ) {
                for ( unsigned int m{i + 1}; m <= N; ++m ) {
                   --a_i[m];
                }

                for ( unsigned int n{k}; n < a_i[N]; ++n ) {
                    a_j[n] = a_j[n + 1];
                    a_entries[n] = a_entries[n + 1];
                }

                return;
            } else if ( a_j[k] > j ) {
                return;
            }
        }
    } else {
        unsigned int k{a_i[i]};

        for ( ; k < a_i[i + 1]; ++k ) {
            if ( a_j[k] == j ) {
                a_entries[n] = s;
                return;
            } else if ( a_j[k] > j ) {
                break;
            }
        }

        if ( off_capacity == a_i[N] ) {
            resize( a_i[N] + 10 );
        }

        for ( unsigned int n{a_i[N]}; n > k; --n ) {
            a_j[n] = a_j[n - 1];
            a_entries[n] = a_entries[n - 1];
        }

        for ( unsigned int m{i + 1}; m <= N; ++m ) {
             ++a_i[m];
        }

        a_j[k] = j;
        a_entries[k] = s;
    }
}
```

There are three types of matrix-vector multiplication $A\mathbf{v}$ that are common:

1. full matrix multiplication,
2. a multiplication of only the diagonal entries, and
3. a multiplication of only the off-diagonal entries.

The user can specify each of these.

```
template <unsigned int N>
vec<N> matrix::multiply( vec<N> v, multiplication_t type = FULL ) {
    vec<N> result;

    for ( unsigned int i{0}; i < N; ++i ) {
        result[i] = ( type == OFF_DIAGONAL )? 0.0 : a_diagonals[i]*v[i];

        if ( type != OFF_DIAGONAL ) {
            for ( unsigned int j{a_i[i]}; j < a_i[i + 1]; ++j ) {
                result[i] += a_entries[j]*v[a_j[j]];
            }
        }
    }

    return result;
}
```

Multiplying a matrix by a scalar requires us multiply all of the diagonal entries, and then to either:

1. set the remainder of the matrix to zero if the scalar is 0, or
2. multiply each of the off-diagonal entries by that scalar.

In theory, the product of two non-zero double-precision floating-point numbers may equal zero, in which case they should be removed from the list of non-zero entries; however, this is likely rare.

```
template <unsigned int N>
matrix<N> &matrix<N>::operator *=( double s ) {
    for ( unsigned int i{0}; i < N; ++i ) {
        a_diagonals[i] *= s;
    }

    if ( s == 0.0 ) {
        for ( unsigned int i{0}; i <= N; ++i ) {
            a_i[i] = 0;
        }
    } else {
        // This does note check if an off-diagonal entry times 's' equals 0.0
        for ( unsigned int j{0}; j < a_i[N]; ++j ) {
            a_entries[j] *= s;
        }
    }

    return *this;
}

template <unsigned int N>
matrix<N> matrix<N>::operator *( double s ) {
    matrix<N> matrix{ *this };

    matrix *= s;

    return matrix;
}
```

Problems

1. Given the following matrix, show its implementation using the above-described sparse matrix data structure.

$$\begin{pmatrix} 3.2 & 0 & 0 & 0.3 & 0 & 0 & 0 & 0 \\ 0 & 7.4 & 0 & 0 & 0.2 & 0 & 0.6 & 0 \\ 0 & 0 & -5.9 & 0 & 0 & 0 & 0 & 0 \\ 0.1 & 0 & 0 & 8.0 & 0 & 0 & -0.5 & 0 \\ 0 & -0.4 & 0 & 0 & -6.1 & 0 & 0 & 0.7 \\ 0 & 0 & 0 & 0 & 0 & 4.5 & 0 & 0 \\ 0 & 0.6 & 0 & -0.9 & 0 & 0 & -9.3 & 0 \\ 0 & 0 & 0 & 0 & 0.8 & 0 & 0 & 1.8 \end{pmatrix}$$

2. Given the following matrix, show its implementation using the above-described sparse matrix data structure:

$$\begin{pmatrix} 3.2 & 0.3 & 0 & 0 & 0 & 0 & 0.7 & 0 \\ 0 & 7.4 & 0.8 & 0 & 0 & 0 & 0 & 0.2 \\ 0 & 0 & -5.9 & 0.4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8.0 & 0.1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -6.1 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4.5 & 0.9 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -9.3 & 0.6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.8 \end{pmatrix}$$

3. Why is it likely less efficient to use a binary search on a very small sub-array as opposed to a more straight-forward linear search?

4. The above scalar multiplication implementation assumes that a scalar multiplied by an entry is non-zero; however, it is entirely possible for the product of two double-precision floating-point numbers to be zero even when the operands themselves are both non-zero. For example, `1e-162` when multiplied by itself produces `0`. How would you have to rewrite the implementation if you did not want to include numbers such that $sa_{i,j} = 0$ even if neither s nor $a_{i,j}$ are zero?

Acknowledgments