Thus, upon substitution in the BVP we obtain

$$\frac{1}{\Delta x^2}(\theta_{i-1} - 2\theta_i + \theta_{i+1}) = \frac{1}{12}(q_{i-1} + 10q_i + q_{i+1}) + \mathcal{O}(\Delta x^4).$$

This rather simple example helps us to understand what is the difference between the *explicit* and the *implicit* discretization. Specifically, here following the implicit discretization we have effectively modified the right-hand-side (RHS) through a *mass matrix*, that is we have distributed the forcing around the node of interest, just like in finite element methods (e.g., see [59]). Therefore, for the RHS we have:

$$\text{RHS} : \frac{1}{12}\underbrace{\begin{bmatrix} & \ddots & \ddots & \text{O} \\ & 1 & 10 & 1 \\ \text{O} & & \ddots & \ddots \end{bmatrix}}_{\text{Mass Matrix}} \begin{bmatrix} \vdots \\ q_{i-1} \\ q_i \\ q_{i+2} \\ \vdots \end{bmatrix}$$

Note that what we have obtained by expanding the RHS in this particular BVP is identical to the most compact scheme of $\mathcal{O}(\Delta x^4)$ for $\alpha = 1/10$, i.e.,

$$\frac{1}{10}(\underbrace{\theta_{xx}}_{q_{i-1}})_{i-1} + (\underbrace{\theta_{xx}}_{q_i})_i + \frac{1}{10}(\underbrace{\theta_{xx}}_{q_{i+1}})_{i+1} = \frac{12}{10}\frac{\theta_{i+1} - 2\theta_i + \theta_{i+1}}{\Delta x^2} + \mathcal{O}(\Delta x^4).$$

This formula is exactly what we have derived previously in equation (6.4).

6.1.4 Thomas Algorithm for Tridiagonal Systems

Three-point stencils lead to second-order accuracy and fourth-order accuracy for explicit and implicit discretizations of second-order boundary value problems, respectively. As we have seen from the example above, solution of such BVPs reduces to solving the linear system

$$\mathbf{A}\,\mathbf{x} = \mathbf{q}\,,$$

where the matrix $\mathbf{A}$ is tridiagonal if the boundary conditions are Dirichlet. In the following, we demonstrate how to solve this system using the *Thomas algorithm*, which is a special case of Gaussian elimination, see also section 9.1. This method consists of three main steps:

- The LU decomposition of matrix $\mathbf{A}$, that is its factorization into a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$. Note that this factorization maintains the bandwidth, and therefore both matrices $\mathbf{L}$ and $\mathbf{U}$ are bidiagonal.
- The forward substitution where the matrix $\mathbf{L}$ is involved, and
- The final backward substitution, where the matrix $\mathbf{U}$ is involved.

More specifically, we have:

$$\underbrace{\begin{bmatrix} a_1 & c_1 & & & \\ b_2 & a_2 & c_2 & 0 & \\ & b_3 & a_3 & c_3 & \\ & \ddots & \ddots & \ddots & \\ & 0 & & b_N & a_N \end{bmatrix}}_{\mathbf{A}} = \underbrace{\begin{bmatrix} 1 & & & & \\ \ell_2 & 1 & & 0 & \\ & \ell_3 & 1 & & \\ & & \ddots & \ddots & \\ & 0 & & \ell_N & 1 \end{bmatrix}}_{\mathbf{L}} \underbrace{\begin{bmatrix} d_1 & u_1 & & & \\ & d_2 & u_2 & 0 & \\ & & d_3 & u_3 & \\ & & 0 & \ddots & \ddots \\ & & & & d_N \end{bmatrix}}_{\mathbf{U}}$$

Step 1: (LU Decomposition) $\mathbf{A} = \mathbf{LU}$.

We determine the elements of matrices $\mathbf{L}$ and $\mathbf{U}$ in three stages, separating the end-points from the interior points as follows:

$$d_1 = a_1, \quad u_1 = c_1$$

$$i^{\text{th}} \begin{cases} \ell_i d_{i-1} = b_i \Rightarrow \ell_i = b_i/\ell_i d_{i-1}, & (N \textit{ multiplications}) \\ \ell_i u_{i-1} + d_i = a_i \Rightarrow d_i = a_i - \ell_i u_{i-1} & (N \textit{ multiplications}, N \textit{ additions}) \\ u_i = c_i \end{cases}$$

$$N^{\text{th}} \begin{cases} \ell_N d_{N-1} = b_N \Rightarrow \ell_N = b_N/d_{N-1} \\ \ell_N u_{N-1} + d_N = a_N \Rightarrow d_N = a_N - \ell_N u_{N-1} \end{cases}$$

Therefore, from the above we see that the total *computational complexity* for an LU decomposition of a tridiagonal matrix corresponds to $2N$ multiplications and N additions.

Step 2: (Forward Substitution) $\mathbf{Ly} = \mathbf{q}$ The intermediate vector $\mathbf{y}$ is determined from

$$\begin{bmatrix} 1 & & & & \\ \ell_2 & 1 & & & \\ & \ell_3 & 1 & & \\ & & \ddots & \ddots & \\ & & & \ell_N & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ \vdots \\ q_N \end{bmatrix} \Rightarrow \begin{cases} y_1 = q_1 \\ \ell_i y_{i-1} + y_i = q_i \Rightarrow y_i = q_i - \ell_i y_{i-1} \end{cases}$$

Here the operation count is N multiplications and N additions.

Step 3: (Backward Substitution) $\mathbf{Ux} = \mathbf{y}$

In the final step we have

$$\begin{bmatrix} d_1 & u_1 & & & \\ & d_2 & u_2 & & \\ & & d_3 & u_3 & \\ & 0 & & \ddots & \ddots \\ & & & & d_N \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_N \end{bmatrix}$$

and the final solution is obtained from:

$$\begin{cases} x_N = y_N/d_N \\ d_i x_i + u_i x_{i+1} = y_i \Rightarrow x_i = (y_i - u_i x_{i+1})/d_i, \quad i = N-1, \ldots, 1 \end{cases}$$

The corresponding operation count is $2N$ multiplications and N additions.

We can now summarize the operation count:

1) LU:	$2N$ multiplications,	N additions
2) Forward:	N multiplications,	N additions
3) Backward:	$2N$ multiplications,	N additions
Total:	$5N$ multiplications,	$3N$ additions

Remark: It can be shown that the above algorithm will always converge if the tridiagonal system is diagonally dominant, i.e.,

$$\begin{aligned} |a_k| &\geq |b_k| + |c_k|, \quad k = 2, \ldots, N-1 \\ |a_1| &> |c_1| \text{ and } |a_N| > |b_N| \,. \end{aligned}$$

Also, if a, b, c are matrices instead of scalars we will have a *block-tridiagonal* system and the same algorithm can be applied.

WARNING

Programmer Beware!

- Think carefully about indexing. Simple mistakes can cause major headaches!

Software ⊙ *Suite* *Putting it into Practice*

Below we present a serial C++ implementation of the Thomas algorithm presented above.

```
void ThomasAlgorithm(int N, double *b, double *a, double *c,
                     double *x, double *q){
  int i;
  double *l,*u,*d,*y;

  l = new double[N];
  u = new double[N];
  d = new double[N];
```

```
  y = new double[N];

  /* LU Decomposition */
  d[0] = a[0];
  u[0] = c[0];
  for(i=0;i<N-2;i++){
    l[i] = b[i]/d[i];
    d[i+1] = a[i+1] - l[i]*u[i];
    u[i+1] = c[i+1];
  }
  l[N-2] = b[N-2]/d[N-2];
  d[N-1] = a[N-1] - l[N-2]*u[N-2];

  /* Forward Substitution [L][y] = [q] */
  y[0] = q[0];
  for(i=1;i<N;i++)
    y[i] = q[i] - l[i-1]*y[i-1];

  /* Backward Substitution [U][x] = [y] */
  x[N-1] = y[N-1]/d[N-1];
  for(i=N-2;i>=0;i--)
    x[i] = (y[i] - u[i]*x[i+1])/d[i];

  delete[] l;
  delete[] u;
  delete[] d;
  delete[] y;
  return;
}
```

Common Programming Trick: Notice that every time that we call the above routine we must *allocate* and *deallocate* memory. Suppose that we are calling this routine over and over, and using the same size allocation each time. We are wasting a lot of time just allocating and deallocating! What can be done? One common trick is to use *static*. When a variable is declared static, it is allocated each once (in the static part of a program's memory), and remains throughout the duration of the program. Hence, if you declare a pointer variable as static within the routine, and allocate an array of memory the first time that the routine is called, then you can dispense with allocating/deallocating each time. This is demonstrated in the following modified code.

```
void ThomasAlgorithm(int N, double *b, double *a, double *c,
                     double *x, double *q){
  int i;
  static double *l=NULL,*u=NULL,*d=NULL,*y=NULL;
```

```
  if(l == NULL){
    l = new double[N];
    u = new double[N];
    d = new double[N];
    y = new double[N];
  }

  /* LU Decomposition */
  d[0] = a[0];
  u[0] = c[0];
  for(i=0;i<N-2;i++){
    l[i] = b[i]/d[i];
    d[i+1] = a[i+1] - l[i]*u[i];
    u[i+1] = c[i+1];
  }
  l[N-2] = b[N-2]/d[N-2];
  d[N-1] = a[N-1] - l[N-2]*u[N-2];

  /* Forward Substitution [L][y] = [q] */
  y[0] = q[0];
  for(i=1;i<N;i++)
    y[i] = q[i] - l[i-1]*y[i-1];

  /* Backward Substitution [U][x] = [y] */
  x[N-1] = y[N-1]/d[N-1];
  for(i=N-2;i>=0;i--)
    x[i] = (y[i] - u[i]*x[i+1])/d[i];

  return;
}
```

Remark: In the example we show above, the pointer variables are declared static, and are initialized to NULL *the first time that the routine is run.* Since the pointer is NULL, the memory is allocated the first time the routine is run; however, for all subsequent calls, the value of the pointer l is not NULL (it contains some address value), and hence memory is not allocated. Of course, in this implementation we have assumed that the value of N is always less than or equal to the first value of N passed to this routine. More complex schemes can be devised to make allocate/deallocate only when the size changes. This methodology is a common trick - valid with respect to the language, but despised by many as unclean programming!

Key Concept

- You need not recompute things that do not change!

Instead of using static allocations, one preferred way of increasing code re-use is to move the memory allocation outside of the Thomas Algorithm routines and to break the algorithm into two functions:

1. *ThomasAlgorithmLU* - accomplishes the LU decomposition of the matrix **A**. This routine needs to be called only once per matrix **A**.
2. *ThomasAlgorithmSolve* - accomplishes the forward and back substitution. This routine needs to be called every time the right-hand-side value **b** changes.

The memory allocation is moved outside of these functions; the calling function is responsible for memory allocation. We now present both the functions described above.

```
void ThomasAlgorithmLU(int N, double *b, double *a, double *c,
                       double *l, double *u, double *d){
  int i;

  /* LU Decomposition */
  d[0] = a[0];
  u[0] = c[0];
  for(i=0;i<N-2;i++){
    l[i] = b[i]/d[i];
    d[i+1] = a[i+1] - l[i]*u[i];
    u[i+1] = c[i+1];
  }
  l[N-2] = b[N-2]/d[N-2];
  d[N-1] = a[N-1] - l[N-2]*u[N-2];

  return;
}

void ThomasAlgorithmSolve(int N, double *l, double *u, double *d,
                          double *x, double *q){
  int i;
  double *y = new double[N];

  /* Forward Substitution [L][y] = [q] */
  y[0] = q[0];
  for(i=1;i<N;i++)
```

```
    y[i] = q[i] - l[i-1]*y[i-1];

  /* Backward Substitution [U][x] = [y] */
  x[N-1] = y[N-1]/d[N-1];
  for(i=N-2;i>=0;i--)
    x[i] = (y[i] - u[i]*x[i+1])/d[i];

  delete[] y;
  return;
}
```

Remark: Notice that the function which accomplishes the forward solve and the back solve does not require the matrix arrays a, b and c; it requires only the l, u and d arrays which contain the LU decomposition of $\mathbf{A}$. Once the LU decomposition has been accomplished, and if the matrix $\mathbf{A}$ is not needed for any other purpose, the arrays a, b, and c can be deallocated.

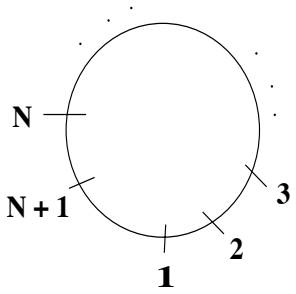

Figure 6.12: Domain for solving the steady heat equation in a ring.

Thomas Algorithm for Periodic Tridiagonal Systems

The boundary value problem we considered in the example above employed Dirichlet boundary conditions, but often *periodic boundary conditions* are required. This could be a case for example, where an infinite domain is simulated or the physics of the problem dictates it, as in solving an elliptic problem on a ring (see figure 6.12). In this case, despite the sparsity of the matrix resulting from the discretization and its almost tridiagonal form everywhere, the bandwidth is actually equal to the order of the matrix in the form shown below:

$$\begin{bmatrix} a_1 & c_1 & & & b_1 \\ b_2 & a_2 & c_2 & 0 & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & c_N \\ c_{N+1} & 0 & & b_{N+1} & a_{N+1} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \\ x_{N+1} \end{bmatrix} = \begin{bmatrix} q \end{bmatrix}$$

where we assume that b_1 and c_{N+1} are coefficients corresponding to the periodic boundary conditions, (e.g., equal to 1 in the example above).

We can solve this system by first "condensing" the matrix, that is eliminating the last row and the last column, to arrive at:

$$\underbrace{\begin{bmatrix} a_1 & c_1 & & & \\ b_2 & a_2 & c_2 & 0 & \\ & \ddots & \ddots & & \\ & & & & c_{N-1} \\ 0 & & & b_N & a_N \end{bmatrix}}_{\mathbf{A}^c} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \\ x_N \end{bmatrix} = \mathbf{q} - \begin{bmatrix} b_1 \\ 0 \\ \vdots \\ 0 \\ c_N \end{bmatrix} x_{N+1} \, .$$

Now we use the linear property and propose a superposition of the form

$$\mathbf{x} = \mathbf{x}^{(1)} + \mathbf{x}^{(2)} \cdot x_{N+1} \, ,$$

where $x^{(1)}$ and $x^{(2)}$ are solutions of the tridiagonal "condensed" system with N unknowns, i.e.,

$$\begin{aligned} \mathbf{A}^c \mathbf{x}^{(1)} &= \mathbf{q} \\ \mathbf{A}^c \mathbf{x}^{(2)} &= \begin{bmatrix} -b_1 \\ 0 \\ \vdots \\ 0 \\ -c_N \end{bmatrix} . \end{aligned}$$

We finally compute x_{N+1} from the last equation in the original system by back substitution, i.e.,

$$c_{N+1}(x_1^{(1)} + x_{N+1}x_1^{(2)}) + b_{N+1}(x_N^{(1)} + x_{N+1}x_N^{(2)}) + a_{N+1}x_{N+1} = q_{N+1}$$

and we solve for x_{N+1}:

$$x_{N+1} = \frac{q_{N+1} - c_{N+1}x_1^{(1)} - b_{N+1}x_N^{(1)}}{a_{N+1} + c_{N+1}x_1^{(2)} + b_{N+1}x_N^{(2)}} \, .$$

Software  *Suite* *Putting it into Practice*

Below we present a serial C++ implementation of the Thomas Algorithm for *periodic systems*. As was discussed above, the Thomas Algorithm for periodic systems requires us to accomplish LU solves on *condensed systems*. Note that we accomplish this by re-using the Thomas algorithm functions that we previously presented on the condensed system.

```
void ThomasAlgorithm_per(int N, double *b, double *a, double *c,
                         double *x, double *q){
  int i;
  double *x1,*x2,*q2;

  x1 = new double[N-1];
  x2 = new double[N-1];
  q2 = new double[N-1];

  /* Prepare secondary q */
  for(i=0;i<N-1;i++)
    q2[i] = 0.0;
  q2[0] = -b[N-1];
  q2[N-2] = -c[N-2];

  ThomasAlgorithm(N-1,b,a,c,x1,q);
  ThomasAlgorithm(N-1,b,a,c,x2,q2);

  x[N-1] = (q[N-1] - c[N-1]*x1[0] - b[N-2]*x1[N-2])/
    (a[N-1] + c[N-1]*x2[0] + b[N-2]*x2[N-2]);

  for(i=0;i<N-1;i++)
    x[i] = x1[i] + x2[i]*x[N-1];

  delete[] x1;
  delete[] x2;
  delete[] q2;
}
```

Key Concept

- Code re-use is important. If you have already invested the time to make sure that a routine is well-written and correctly implemented, then you can use the routine as a component in new routines.

6.1.5 Parallel Algorithm for Tridiagonal Systems

In seeking a parallelization strategy for solving triagonal systems, we will once again examine the structure of the LU decomposition as we did in formulating the Thomas algorithm. By exploiting the recursive nature of the LU decomposition, we will devise a *full-recursive-doubling*