we find that triangular matrix multiplication requires one-sixth the number of flops as full matrix multiplication:

$$\sum_{i=1}^{n}\sum_{j=i}^{n} 2(j-i+1) = \sum_{i=1}^{n}\sum_{j=1}^{n-i+1} 2j \approx \sum_{i=1}^{n} \frac{2(n-i+1)^2}{2} = \sum_{i=1}^{n} i^2 \approx \frac{n^3}{3}.$$

We throw away the low-order terms since their inclusion does not contribute to what the flop count "says." For example, an exact flop count of Algorithm 1.2.1 reveals that precisely $n^3/3 + n^2 + 2n/3$ flops are involved. For large $n$ (the typical situation of interest) we see that the exact flop count offers no insight beyond the simple $n^3/3$ accounting.

Flop counting is a necessarily crude approach to the measurement of program efficiency since it ignores subscripting, memory traffic, and other overheads associated with program execution. We must not infer too much from a comparison of flop counts. We cannot conclude, for example, that triangular matrix multiplication is six times faster than full matrix multiplication. Flop counting captures just one dimension of what makes an algorithm efficient in practice. The equally relevant issues of vectorization and data locality are taken up in §1.5.

### 1.2.5 Band Storage

Suppose $A \in \mathbb{R}^{n\times n}$ has lower bandwidth $p$ and upper bandwidth $q$ and assume that $p$ and $q$ are much smaller than $n$. Such a matrix can be stored in a $(p+q+1)$-by-$n$ array $A.band$ with the convention that

$$a_{ij} = A.band(i-j+q+1, j) \tag{1.2.1}$$

for all $(i,j)$ that fall inside the band, e.g.,

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & a_{35} & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} & a_{46} \\ 0 & 0 & 0 & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & 0 & a_{65} & a_{66} \end{bmatrix} \Rightarrow \begin{bmatrix} * & * & a_{13} & a_{24} & a_{35} & a_{46} \\ * & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \\ a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & * \end{bmatrix}.$$

Here, the "*" entries are unused. With this data structure, our column-oriented gaxpy algorithm (Algorithm 1.1.4) transforms to the following:

**Algorithm 1.2.2 (Band Storage Gaxpy)** Suppose $A \in \mathbb{R}^{n\times n}$ has lower bandwidth $p$ and upper bandwidth $q$ and is stored in the $A.band$ format (1.2.1). If $x, y \in \mathbb{R}^n$, then this algorithm overwrites $y$ with $y + Ax$.

```
for $j = 1{:}n$
    $\alpha_1 = \max(1, j-q),\ \alpha_2 = \min(n, j+p)$
    $\beta_1 = \max(1, q+2-j),\ \beta_2 = \beta_1 + \alpha_2 - \alpha_1$
    $y(\alpha_1{:}\alpha_2) = y(\alpha_1{:}\alpha_2) + A.band(\beta_1{:}\beta_2, j)x(j)$
end
```

Notice that by storing $A$ column by column in *A.band*, we obtain a column-oriented saxpy procedure. Indeed, Algorithm 1.2.2 is derived from Algorithm 1.1.4 by recognizing that each saxpy involves a vector with a small number of nonzeros. Integer arithmetic is used to identify the location of these nonzeros. As a result of this careful zero/nonzero analysis, the algorithm involves just $2n(p+q+1)$ flops with the assumption that $p$ and $q$ are much smaller than $n$.

## 1.2.6 Working with Diagonal Matrices

Matrices with upper and lower bandwidth zero are *diagonal*. If $D \in \mathbb{R}^{m \times n}$ is diagonal, then we use the notation

$$D = \text{diag}(d_1, \ldots, d_q), \quad q = \min\{m,n\} \quad \Longleftrightarrow \quad d_i = d_{ii}.$$

Shortcut notations when the dimension is clear include diag($d$) and diag($d_i$). Note that if $D = \text{diag}(d) \in \mathbb{R}^{n \times n}$ and $x \in \mathbb{R}^n$, then $Dx = d.*x$. If $A \in \mathbb{R}^{m \times n}$, then pre-multiplication by $D = \text{diag}(d_1, \ldots, d_m) \in \mathbb{R}^{m \times m}$ scales rows,

$$B = DA \quad \Longleftrightarrow \quad B(i,:) = d_i \cdot A(i,:),\ i = 1{:}m$$

while post-multiplication by $D = \text{diag}(d_1, \ldots, d_n) \in \mathbb{R}^{n \times n}$ scales columns,

$$B = AD \quad \Longleftrightarrow \quad B(:,j) = d_j \cdot A(:,j),\ j = 1{:}n.$$

Both of these special matrix-matrix multiplications require $mn$ flops.

## 1.2.7 Symmetry

A matrix $A \in \mathbb{R}^{n \times n}$ is *symmetric* if $A^T = A$ and *skew-symmetric* if $A^T = -A$. Likewise, a matrix $A \in \mathbb{C}^{n \times n}$ is *Hermitian* if $A^H = A$ and *skew-Hermitian* if $A^H = -A$. Here are some examples:

$$\text{Symmetric:} \quad \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix}, \qquad \text{Hermitian:} \quad \begin{bmatrix} 1 & 2-3i & 4-5i \\ 2+3i & 6 & 7-8i \\ 4+5i & 7+8i & 9 \end{bmatrix},$$

$$\text{Skew-Symmetric:} \quad \begin{bmatrix} 0 & -2 & 3 \\ 2 & 0 & -5 \\ -3 & 5 & 0 \end{bmatrix}, \quad \text{Skew-Hermitian:} \quad \begin{bmatrix} i & -2+3i & -4+5i \\ 2+3i & 6i & -7+8i \\ 4+5i & 7+8i & 9i \end{bmatrix}.$$

For such matrices, storage requirements can bc halved by simply storing the lower triangle of elements, e.g.,

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix} \quad \Leftrightarrow \quad A.vec = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}.$$

For general $n$, we set

$$A.vec((n - j/2)(j-1) + i) \; = \; a_{ij} \quad 1 \le j \le i \le n. \tag{1.2.2}$$