# II.1 Structured Matrices

We have seen how algebraic operations (`+`, `-`, `*`, `/`) are defined exactly in terms of rounding ($\oplus$, $\ominus$, $\otimes$, $\oslash$) for floating point numbers. Now we see how this allows us to do (approximate) linear algebra operations on matrices.

A matrix can be stored in different formats. Here we consider the following structures:

1. *Dense*: This can be considered unstructured, where we need to store all entries in a

vector or matrix. Matrix multiplication reduces directly to standard algebraic operations. Solving linear systems with dense matrices will be discussed later. 2. *Triangular*: If a matrix is upper or lower triangular, we can immediately invert using back-substitution. In practice we store a dense matrix and ignore the upper/lower entries. 3. *Banded*: If a matrix is zero apart from entries a fixed distance from the diagonal it is called banded and this allows for more efficient algorithms. We discuss diagonal, tridiagonal and bidiagonal matrices.

In the next chapter we consider more complicated orthogonal matrices.

In [1]:
```
# LinearAlgebra contains routines for doing linear algebra
# BenchmarkTools is a package for reliable timing
using LinearAlgebra, Plots, BenchmarkTools, Test
```

---

## 1. Dense vectors and matrices

A `Vector` of a primitive type (like `Int` or `Float64`) is stored consecutively in memory: that is, a vector consists of a memory address (a *pointer*) to the first entry and a length. E.g. if we have a `Vector{Int8}` of length `n` then it is stored as `8n` bits (`n` bytes) in a row. That is, if the memory address of the first entry is `k` and the type is `T`, the memory address of the second entry is `k + sizeof(T)`.

---

**Remark (advanced)** We can actually experiment with this (NEVER DO THIS IN PRACTICE!!), beginning with an 8-bit type:

In [2]:
```
a = Int8[2, 4, 5]
p = pointer(a) # pointer(a) returns memory address of the first entry, which
# We can think of a pointer as simply a UInt64 alongside a Type to interpret
```

Out[2]:
```
Ptr{Int8} @0x000000014539fd78
```

We can see what's stored at a pointer as follows:

In [3]:
```
Base.unsafe_load(p) # loads data at `p`. Knows its an `Int8` because of type
```
Out[3]:
```
2
```

Adding an integer to a pointer gives a new pointer with the address incremented:

In [4]:
```
p + 1 # memory address of next entry, which is 1 more than first
```
Out[4]:
```
Ptr{Int8} @0x000000014539fd79
```

We see that this gives us the next entry:

In [5]:
```
Base.unsafe_load(p) # loads data at `p+1`, which is second entry of the vect
```
Out[5]:
```
2
```

For other types we need to increment the address by the size of the type:

In [6]:
```
a = [2.0, 1.3, 1.4]
p = pointer(a)
Base.unsafe_load(p + 8) # sizeof(Float64) == 8
```
Out[6]:
```
1.3
```

Why not do this in practice? It's unsafe because there's nothing stopping us from going past the end of an array:

In [7]:
```
Base.unsafe_load(p + 3 * 8) # whatever bits happened to be next in memory, u
```
Out[7]:
```
2.405888678e-314
```

This may even crash Julia! (I got lucky that it didn't when producing the notes.)

---

A `Matrix` is stored consecutively in memory, going down column-by- column (*column-major*). That is,

In [8]:
```
A = [1 2;
     3 4;
     5 6]
```
Out[8]:
```
3×2 Matrix{Int64}:
 1  2
 3  4
 5  6
```

Is actually stored equivalently to a length `6` vector:

In [9]:
```
vec(A)
```

```
Out[9]: 6-element Vector{Int64}:
 1
 3
 5
 2
 4
 6
```

which in this case would be stored using in `8 * 6 = 48` consecutive memory addresses. That is, a matrix is a pointer to the first entry alongside two integers dictating the row and column sizes.

---

**Remark (advanced)** Note that transposing `A` is done lazyily and so `transpose(A)` (which is equivalent to the adjoint/conjugate-transpose `A'` when the entries are real), is just a special type with a single field: `transpose(A).parent == A`. This is equivalent to *row-major* format, where the next address in memory of `transpose(A)` corresponds to moving along the row.

---

Matrix-vector multiplication works as expected:

```
In [10]: x = [7, 8]
         A * x
```

```
Out[10]: 3-element Vector{Int64}:
 23
 53
 83
```

Note there are two ways this can be implemented:

**Algorithm 1 (matrix-vector multiplication by rows)** For a ring $R$ (typically $\mathbb{R}$ or $\mathbb{C}$), $A \in R^{m\times n}$ and $\mathbf{x} \in R^n$ we have

$$A\mathbf{x} = \begin{bmatrix} \sum_{j=1}^n a_{1,j}x_j \\ \vdots \\ \sum_{j=1}^n a_{m,j}x_j \end{bmatrix}.$$

In code this can be implemented for any types that support `*` and `+` as follows:

```
In [11]: function mul_rows(A, x)
             m,n = size(A)
             # promote_type type finds a type that is compatible with both types, elt
             T = promote_type(eltype(x), eltype(A))
             c = zeros(T, m) # the returned vector, begins of all zeros
             for k = 1:m, j = 1:n
                 c[k] += A[k, j] * x[j] # equivalent to c[k] = c[k] + A[k, j] * x[j]
             end
             c
         end
```

Out[11]: mul_rows (generic function with 1 method)

**Algorithm 2 (matrix-vector multiplication by columns)** For a ring $R$ (typically $\mathbb{R}$ or $\mathbb{C}$ ), $A \in R^{m\times n}$ and $\mathbf{x} \in R^n$ we have

$$A\mathbf{x} = x_1\mathbf{a}_1 + \cdots + x_n\mathbf{a}_n$$

where $\mathbf{a}_j := A\mathbf{e}_j \in R^m$ (that is, the $j$-th column of $A$). In code this can be implemented for any types that support `*` and `+` as follows:

In [12]:
```
function mul_cols(A, x)
    m,n = size(A)
    # promote_type type finds a type that is compatible with both types, elt
    T = promote_type(eltype(x),eltype(A))
    c = zeros(T, m) # the returned vector, begins of all zeros
    for j = 1:n, k = 1:m
        c[k] += A[k, j] * x[j] # equivalent to c[k] = c[k] + A[k, j] * x[j]
    end
    c
end
```

Out[12]: mul_cols (generic function with 1 method)

Both implementations match exactly for integer inputs:

In [13]:
```
mul_rows(A, x), mul_cols(A, x) # also matches `A*x`
```

Out[13]: ([23, 53, 83], [23, 53, 83])

Either implementation will be $O(mn)$ operations. However, the implementation `mul_cols` accesses the entries of `A` going down the column, which happens to be *significantly faster* than `mul_rows`, due to accessing memory of `A` in order. We can see this by measuring the time it takes using `@btime`:

In [14]:
```
n = 1000
A = randn(n,n) # create n x n matrix with random normal entries
x = randn(n) # create length n vector with random normal entries

@btime mul_rows(A,x)
@btime mul_cols(A,x)
@btime A*x; # built-in, high performance implementation. USE THIS in practic
```

```
  1.667 ms (1 allocation: 7.94 KiB)
  755.646 μs (1 allocation: 7.94 KiB)
  220.887 μs (1 allocation: 7.94 KiB)
```

Here `ms` means milliseconds ( `0.001 = 10^(-3)` seconds) and `μs` means microseconds ( `0.000001 = 10^(-6)` seconds). So we observe that `mul` is roughly 3x faster than `mul_rows`, while the optimised `*` is roughly 5x faster than `mul`.

---

**Remark (advanced)** For floating point types, `A*x` is implemented in BLAS which is generally multi-threaded and is not identical to `mul_cols(A,x)`, that is, some inputs will differ in how the computations are rounded.

---

Note that the rules of floating point arithmetic apply here: matrix multiplication with floats will incur round-off error (the precise details of which are subject to the implementation):

```
In [15]: A = [1.4 0.4;
              2.0 1/2]
         A * [1, -1] # First entry has round-off error, but 2nd entry is exact
```

```
Out[15]: 2-element Vector{Float64}:
          0.9999999999999999
          1.5
```

And integer arithmetic will be subject to overflow:

```
In [16]: A = fill(Int8(2^6), 2, 2) # make a matrix whose entries are all equal to 2^6
         A * Int8[1,1] # we have overflowed and get a negative number -2^7
```

```
Out[16]: 2-element Vector{Int8}:
          -128
          -128
```

Solving a linear system is done using `\`:

```
In [17]: A = [1 2 3;
              1 2 4;
              3 7 8]
         b = [10; 11; 12]
         A \ b
```

```
Out[17]: 3-element Vector{Float64}:
           41.000000000000036
          -17.000000000000014
            1.0
```

Despite the answer being integer-valued, here we see that it resorted to using floating point arithmetic, incurring rounding error. But it is "accurate to (roughly) 16-digits". As we shall see, the way solving a linear system works is we first write `A` as a product of matrices that are easy to invert, e.g., a product of triangular matrices or a product of an orthogonal and triangular matrix.

## 2. Triangular matrices

Triangular matrices are represented by dense square matrices where the entries below the diagonal are ignored:

In [18]:
```
A = [1 2 3;
     4 5 6;
     7 8 9]
U = UpperTriangular(A)
```

Out[18]:
```
3×3 UpperTriangular{Int64, Matrix{Int64}}:
 1  2  3
 ⋅  5  6
 ⋅  ⋅  9
```

We can see that `U` is storing all the entries of `A` in a field called `data`:

In [19]:
```
U.data
```

Out[19]:
```
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9
```

Similarly we can create a lower triangular matrix by ignoring the entries above the diagonal:

In [20]:
```
L = LowerTriangular(A)
```

Out[20]:
```
3×3 LowerTriangular{Int64, Matrix{Int64}}:
 1  ⋅  ⋅
 4  5  ⋅
 7  8  9
```

If we know a matrix is triangular we can do matrix-vector multiplication in roughly half the number of operations by skipping over the entries we know are zero:

**Algorithm 3 (upper-triangular matrix-vector multiplication by columns)**

In [21]:
```
function mul_cols(U::UpperTriangular, x)
    n = size(U,1)
    # promote_type type finds a type that is compatible with both types, elt
    T = promote_type(eltype(x),eltype(U))
    b = zeros(T, n) # the returned vector, begins of all zeros
    for j = 1:n, k = 1:j # k = 1:j instead of 1:m since we know U[k,j] = 0 i
        b[k] += U[k, j] * x[j]
    end
    b
end

x = [10, 11, 12]
# matches built-in *
@test mul_cols(U, x) == U*x
```

Out[21]:
```
Test Passed
```

Moreover, we can easily invert matrices. Consider a simple 3×3 example, which can be solved with `\`:

In [22]:
```
b = [5, 6, 7]
x = U \ b # Excercise: why does this return a float vector?
```

Out[22]:
```
3-element Vector{Float64}:
 2.1333333333333333
 0.26666666666666666
 0.7777777777777778
```

Behind the seens, `\` is doing back-substitution: considering the last row, we have all zeros apart from the last column so we know that `x[3]` must be equal to:

In [23]:
```
b[3] / U[3,3]
```

Out[23]:
```
0.7777777777777778
```

Once we know `x[3]`, the second row states `U[2,2]*x[2] + U[2,3]*x[3] == b[2]`, rearranging we get that `x[2]` must be:

In [24]:
```
(b[2] - U[2,3]*x[3])/U[2,2]
```

Out[24]:
```
0.26666666666666666
```

Finally, the first row states `U[1,1]*x[1] + U[1,2]*x[2] + U[1,3]*x[3] == b[1]` i.e. `x[1]` is equal to

In [25]:
```
(b[1] - U[1,2]*x[2] - U[1,3]*x[3])/U[1,1]
```

Out[25]:
```
2.1333333333333333
```

More generally, we can solve the upper-triangular system using *back-substitution*:

**Algorithm 4 (back-substitution)** Let $\mathbb{F}$ be a field (typically $\mathbb{R}$ or $\mathbb{C}$). Suppose $U \in \mathbb{F}^{n\times n}$ is upper-triangular and invertible. Then for $\mathbf{b} \in \mathbb{F}^n$ the solution $\mathbf{x} \in \mathbb{F}^n$ to $U\mathbf{x} = \mathbf{b}$, that is,

$$\begin{bmatrix} u_{11} & \cdots & u_{1n} \\ & \ddots & \vdots \\ & & u_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

is given by computing $x_n, x_{n-1}, \ldots, x_1$ via:

$$x_k = \frac{b_k - \sum_{j=k+1}^n u_{kj} x_j}{u_{kk}}$$

In code this can be implemented for any types that support `*`, `+` and `/` as follows:

In [26]:
```
# ldiv(U, b) is our implementation of U\b
function ldiv(U::UpperTriangular, b)
    n = size(U,1)
    if length(b) != n
        error("The system is not compatible")
    end

    x = zeros(n)  # the solution vector

    for k = n:-1:1  # start with k=n, then k=n-1, ...
        r = b[k]  # dummy variable
        for j = k+1:n
            r -= U[k,j]*x[j] # equivalent to r = r - U[k,j]*x[j]
        end
        # after this for loop, r = b[k] - ∑_{j=k+1}^n U[k,j]x[j]
        x[k] = r/U[k,k]
    end
    x
end

@test ldiv(U, x) ≈ U\x
```

Out[26]:
```
Test Passed
```

The problem sheet will explore implementing multiplication and forward substitution for lower triangular matrices. The cost of multiplying and solving linear systems with a triangular matrix is $O(n^2)$.

---

## 3. Banded matrices

A *banded matrix* is zero off a prescribed number of diagonals. We call the number of (potentially) non-zero diagonals the *bandwidths*:

**Definition 1 (bandwidths)** A matrix $A$ has *lower-bandwidth* $l$ if $A[k,j] = 0$ for all $k - j > l$ and *upper-bandwidth* $u$ if $A[k,j] = 0$ for all $j - k > u$. We say that it has *strictly lower-bandwidth* $l$ if it has lower-bandwidth $l$ and there exists a $j$ such that $A[j+l,j] \neq 0$. We say that it has *strictly upper-bandwidth* $u$ if it has upper-bandwidth $u$ and there exists a $k$ such that $A[k,k+u] \neq 0$.

### Diagonal

**Definition 2 (Diagonal)** *Diagonal matrices* are square matrices with bandwidths $l = u = 0$.

Diagonal matrices in Julia are stored as a vector containing the diagonal entries:

In [27]:
```
x = [1,2,3]
D = Diagonal(x) # the type Diagonal has a single field: D.diag
```

Out[27]:
```
3×3 Diagonal{Int64, Vector{Int64}}:
 1  ⋅  ⋅
 ⋅  2  ⋅
 ⋅  ⋅  3
```

It is clear that we can perform diagonal-vector multiplications and solve linear systems involving diagonal matrices efficiently (in $O(n)$ operations).

## Bidiagonal

**Definition 3 (Bidiagonal)** If a square matrix has bandwidths $(l, u) = (1, 0)$ it is *lower-bidiagonal* and if it has bandwidths $(l, u) = (0, 1)$ it is *upper-bidiagonal*.

We can create Bidiagonal matrices in Julia by specifying the diagonal and off-diagonal:

In [28]:
```
L = Bidiagonal([1,2,3], [4,5], :L) # the type Bidiagonal has three fields: L
```

Out[28]:
```
3×3 Bidiagonal{Int64, Vector{Int64}}:
 1  ⋅  ⋅
 4  2  ⋅
 ⋅  5  3
```

In [29]:
```
Bidiagonal([1,2,3], [4,5], :U)
```

Out[29]:
```
3×3 Bidiagonal{Int64, Vector{Int64}}:
 1  4  ⋅
 ⋅  2  5
 ⋅  ⋅  3
```

Multiplication and solving linear systems with Bidiagonal systems is also $O(n)$ operations, using the standard multiplications/back-substitution algorithms but being careful in the loops to only access the non-zero entries.

## Tridiagonal

**Definition 4 (Tridiagonal)** If a square matrix has bandwidths $l = u = 1$ it is *tridiagonal*.

Julia has a type `Tridiagonal` for representing a tridiagonal matrix from its sub-diagonal, diagonal, and super-diagonal:

In [30]:
```
T = Tridiagonal([1,2], [3,4,5], [6,7]) # The type Tridiagonal has three fiel
```

Out[30]:
```
3×3 Tridiagonal{Int64, Vector{Int64}}:
 3  6  ⋅
 1  4  7
 ⋅  2  5
```

Tridiagonal matrices will come up in solving second-order differential equations and orthogonal polynomials. We will later see how linear systems involving tridiagonal matrices can be solved in $O(n)$ operations.