Structured Matrices

We have seen how algebraic operations (`+` , `-` , `*` , `/`) are well-defined for floating point numbers. Now we see how this allows us to do (approximate) linear algebra operations on structured matrices. That is, 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.
4. *Permutation*: A permutation matrix permutes the rows of a vector.
5. *Orthogonal*: An orthogonal matrix Q satisfies $Q^\top Q = I$, in other words, they are very easy to invert. We discuss the special cases of simple rotations and reflections.

In [1]:
```
using LinearAlgebra, Plots, BenchmarkTools
```

1. Dense vectors and matrices

A `Vector` of a primitive type (like `Int` or `Float64`) is stored consecutively in memory. E.g. if we have a `Vector{Int8}` of length `n` then it is stored as `8n` bits (`n` bytes) in a row. A `Matrix` is stored consecutively in memory, going down column-by- column. That is,

In [2]:
```
A = [1 2;
     3 4;
     5 6]
```

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

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

In [3]:
```
vec(A)
```

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

This is known as *column-major* format.

Remark Note that transposing `A` is done lazyily and so `A'` stores the entries by row. That is, `A'` is stored in *row-major* format.

Matrix-vector multiplication works as expected:

In [4]:
```
x = [7, 8]
A*x
```

Out[4]:
```
3-element Vector{Int64}:
 23
 53
 83
```

Note there are two ways this can be implemented: either the traditional definition, going row-by-row:

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

or going column-by-column:

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

It is easy to implement either version of matrix-multiplication in terms of the algebraic operations we have learned, in this case just using integer arithmetic:

In [5]:
```
# go row-by-row
function mul_rows(A, x)
    m,n = size(A)
    c = zeros(eltype(x), m) # eltype is the type of the elements of a vector/matrix
    for k = 1:m, j = 1:n
        c[k] += A[k, j] * x[j]
    end
    c
end

# go column-by-column
function mul(A, x)
    m,n = size(A)
    c = zeros(eltype(x), m) # eltype is the type of the elements of a vector/matrix
    for j = 1:n, k = 1:m
        c[k] += A[k, j] * x[j]
    end
    c
end

mul_rows(A, x), mul(A, x)
```

Out[5]:
```
([23, 53, 83], [23, 53, 83])
```

Either implementation will be $O(mn)$ operations. However, the implementation `mul` 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 [6]:
```
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(A,x)
@btime A*x; # built-in, high performance implementation. USE THIS in practice
```

```
  2.895 ms (1 allocation: 7.94 KiB)
  858.813 μs (1 allocation: 7.94 KiB)
  172.372 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(A,x)`, that is, some inputs will differ in how the computations are rounded.

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

In [7]:

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

Out[7]:

```
2-element Vector{Float64}:
 0.9999999999999999
 1.5
```

And integer arithmetic will be prone to overflow:

In [8]:

```
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[8]:

```
2-element Vector{Int8}:
 -128
 -128
```

Solving a linear system is done using `\`:

In [9]:

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

Out[9]:

```
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 simpler matrices, e.g., a product of triangular matrices.

Remark (advanced) For floating point types, `A \ x` is implemented in LAPACK, which like BLAS is generally multi-threaded and in fact different machines may round differently.

2. Triangular matrices

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

In [10]:
```
A = [1 2 3;
     4 5 6;
     7 8 9]
U = UpperTriangular(A)
```

Out[10]:
```
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 [11]:
```
U.data
```

Out[11]:
```
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 [12]:
```
L = LowerTriangular(A)
```

Out[12]:
```
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. Moreover, we can easily invert matrices. Consider a simple 3×3 example, which can be solved with `\`:

In [13]:
```
b = [5,6,7]
x = U \ b
```

Out[13]:
```
3-element Vector{Float64}:
 2.1333333333333333
 0.2666666666666666
 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 [14]:
```
b[3] / U[3,3]
```

Out[14]:
```
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 [15]:
```
(b[2] - U[2,3]*x[3])/U[2,2]
```

Out[15]: 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 [16]:
```
(b[1] - U[1,2]*x[2] - U[1,3]*x[3])/U[1,1]
```

Out[16]:
```
2.1333333333333333
```

More generally, we can solve the upper-triangular system

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

by computing $x_n, x_{n-1}, \ldots, x_1$ by the back-substitution formula:

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

The problem sheet will explore implementing this method, as well as forward substitution for inverting 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 (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 (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 [17]:
```
x = [1,2,3]
D = Diagonal(x)
```

Out[17]:
```
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 (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 [18]:
```
Bidiagonal([1,2,3], [4,5], :L)
```

Out[18]:
```
3×3 Bidiagonal{Int64, Vector{Int64}}:
 1  ⋅  ⋅
 4  2  ⋅
 ⋅  5  3
```

In [19]:
```
Bidiagonal([1,2,3], [4,5], :U)
```

Out[19]:
```
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 (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 [20]:
```
Tridiagonal([1,2], [3,4,5], [6,7])
```

Out[20]:
```
3×3 Tridiagonal{Int64, Vector{Int64}}:
 3  6  ⋅
 1  4  7
 ⋅  2  5
```

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

4. Permutation Matrices

Permutation matrices are matrices that represent the action of permuting the entries of a vector, that is, matrix representations of the symmetric group S_n, acting on $\mathbb{R}^n$. Recall every $\sigma \in S_n$ is a bisection between $\{1,2,\ldots,n\}$ and itself. We can write a permutation σ in *Cauchy notation*:

$$\begin{pmatrix} 1 & 2 & 3 & \cdots & n \\ \sigma_1 & \sigma_2 & \sigma_3 & \cdots & \sigma_n \end{pmatrix}$$

where $\{\sigma_1,\ldots,\sigma_n\} = \{1,2,\ldots,n\}$ (that is, each integer appears precisely once). We denote the *inverse permutation* by σ^{-1}, which can be constructed by swapping the rows of the Cauchy notation and reordering.

We can encode a permutation in vector $\boldsymbol{\sigma} = [\sigma_1, \ldots, \sigma_n]^\top$. This induces an action on a vector (using indexing notation)

$$\mathbf{v}[\boldsymbol{\sigma}] = \begin{bmatrix} v_{\sigma_1} \\ \vdots \\ v_{\sigma_n} \end{bmatrix}$$

Example (permutation of a vector) Consider the permutation σ given by

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 4 & 2 & 5 & 3 \end{pmatrix}$$

We can apply it to a vector:

In [21]:
```
σ = [1, 4, 2, 5, 3]
v = [6, 7, 8, 9, 10]
v[σ] # we permutate entries of v
```

Out[21]:
```
5-element Vector{Int64}:
  6
  9
  7
 10
  8
```

Its inverse permutation σ^{-1} has Cauchy notation coming from swapping the rows of the Cauchy notation of σ and sorting:

$$\begin{pmatrix} 1 & 4 & 2 & 5 & 3 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 2 & 4 & 3 & 5 \\ 1 & 3 & 2 & 5 & 4 \end{pmatrix}$$

Julia has the function `invperm` for computing the vector that encodes the inverse permutation: And indeed:

In [22]:
```
σ⁻¹ = invperm(σ) # note that ⁻¹ are just unicode characters in the variable name
```

Out[22]:
```
5-element Vector{Int64}:
 1
 3
 5
 2
 4
```

And indeed permuting the entries by `σ` and then by `σ⁻¹` returns us to our original vector:

In [23]:
```
v[σ][σ⁻¹] # permuting by σ and then σⁱ gets us back
```

Out[23]:
```
5-element Vector{Int64}:
  6
  7
  8
  9
 10
```

Note that the operator

$$P_\sigma(\mathbf{v}) = \mathbf{v}[\boldsymbol{\sigma}]$$

is linear in $\mathbf{v}$, therefore, we can identify it with a matrix whose action is:

$$P_\sigma \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} v_{\sigma_1} \\ \vdots \\ v_{\sigma_n} \end{bmatrix}.$$

The entries of this matrix are

$$P_\sigma[k,j] = \mathbf{e}_k^\top P_\sigma \mathbf{e}_j = \mathbf{e}_k^\top \mathbf{e}_{\sigma_j^{-1}} = \delta_{k,\sigma_j^{-1}} = \delta_{\sigma_k,j}$$

where $\delta_{k,j}$ is the *Kronecker delta*:

$$\delta_{k,j} := \begin{cases} 1 & k = j \\ 0 & \text{otherwise} \end{cases}.$$

This construction motivates the following definition:

Definition (permutation matrix) $P \in \mathbb{R}^{n \times n}$ is a permutation matrix if it is equal to the identity matrix with its rows permuted.

Example (5×5 permutation matrix) We can construct the permutation representation for σ as above as follows:

In [24]:

```
P = I(5)[σ,:]
```

Out[24]:

```
5×5 SparseMatrixCSC{Bool, Int64} with 5 stored entries:
 1  ⋅  ⋅  ⋅  ⋅
 ⋅  ⋅  ⋅  1  ⋅
 ⋅  1  ⋅  ⋅  ⋅
 ⋅  ⋅  ⋅  ⋅  1
 ⋅  ⋅  1  ⋅  ⋅
```

And indeed, we see its action is as expected:

In [25]:

```
P * v
```

Out[25]:

```
5-element Vector{Int64}:
  6
  9
  7
 10
  8
```

Remark (advanced) Note that `P` is a special type `SparseMatrixCSC`. This is used to represent a matrix by storing only the non-zero entries as well as their location. This is an important data type in high-performance scientific computing, but we will not be using general sparse matrices in this module.

Proposition (permutation matrix inverse) Let P_σ be a permutation matrix corresponding to the permutation σ. Then

$$P_\sigma^\top = P_{\sigma^{-1}} = P_\sigma^{-1}$$

That is, P_σ is *orthogonal*:

$$P_\sigma^\top P_\sigma = P_\sigma P_\sigma^\top = I.$$

Proof

We prove orthogonality via:

$$\mathbf{e}_k^\top P_\sigma^\top P_\sigma \mathbf{e}_j = (P_\sigma \mathbf{e}_k)^\top P_\sigma \mathbf{e}_j = \mathbf{e}_{\sigma_k^{-1}}^\top \mathbf{e}_{\sigma_j^{-1}} = \delta_{k,j}$$

This shows $P_\sigma^\top P_\sigma = I$ and hence $P_\sigma^{-1} = P_\sigma^\top$.

∎

Permutation matrices are examples of sparse matrices that can be very easily inverted.

4. Orthogonal matrices

Definition (orthogonal matrix) A square matrix is *orthogonal* if its inverse is its transpose:

$$Q^\top Q = QQ^\top = I.$$

We have already seen an example of an orthogonal matrices (permutation matrices). Here we discuss two important special cases: simple rotations and reflections.

Simple rotations

Definition (simple rotation) A 2×2 *rotation matrix* through angle θ is

$$Q_\theta := \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

In what follows we use the following for writing the angle of a vector:

Definition (two-arg arctan) The two-argument arctan function gives the angle `θ` through the point $[a, b]^\top$, i.e.,

$$\sqrt{a^2 + b^2} \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix}$$

It can be defined in terms of the standard arctan as follows:

$$\operatorname{atan}(b, a) := \begin{cases} \operatorname{atan}\frac{b}{a} & a > 0 \\ \operatorname{atan}\frac{b}{a} + \pi & a < 0 \text{ and } b > 0 \\ \operatorname{atan}\frac{b}{a} + \pi & a < 0 \text{ and } b < 0 \\ \pi/2 & a = 0 \text{ and } b > 0 \\ -\pi/2 & a = 0 \text{ and } b < 0 \end{cases}$$

This is available in Julia:

In [26]:
```
atan(-1,-2) # angle through [-2,-1]
```

Out[26]:
```
-2.677945044588987
```

We can rotate an arbitrary vector to the unit axis. Interestingly it only requires basic algebraic functions (no trigonometric functions):

Proposition (rotation of a vector) The matrix

$$Q = \frac{1}{\sqrt{a^2+b^2}} \begin{bmatrix} a & b \\ -b & a \end{bmatrix}$$

is a rotation matrix satisfying

$$Q \begin{bmatrix} a \\ b \end{bmatrix} = \sqrt{a^2+b^2} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Proof

The last equation is trivial so the only question is that it is a rotation matrix. Define $\theta = -\operatorname{atan}(b,a)$. By definition of the two-arg arctan we have

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} = \begin{bmatrix} \cos(-\theta) & \sin(-\theta) \\ -\sin(-\theta) & \cos(-\theta) \end{bmatrix} = \frac{1}{\sqrt{a^2+b^2}} \begin{bmatrix} a & b \\ -b & a \end{bmatrix}.$$

∎

Reflections

In addition to rotations, another type of orthognal matrix are reflections:

Definition (reflection matrix) Given a vector $\mathbf{v}$ satisfying $\|\mathbf{v}\| = 1$, the reflection matrix is the orthogonal matrix

$$Q_{\mathbf{v}} \triangleq I - 2\mathbf{v}\mathbf{v}^\top$$

These are reflections in the direction of $\mathbf{v}$. We can show this as follows:

Proposition $Q_{\mathbf{v}}$ satisfies:

1. Symmetry: $Q_{\mathbf{v}} = Q_{\mathbf{v}}^\top$
2. Orthogonality: $Q_{\mathbf{v}}^\top Q_{\mathbf{v}} = I$
3. $\mathbf{v}$ is an eigenvector of $Q_{\mathbf{v}}$ with eigenvalue -1
4. $Q_{\mathbf{v}}$ is a rank-1 perturbation of I
5. $\det Q_{\mathbf{v}} = -1$

Proof

Property 1 follows immediately. Property 2 follows from

$$Q_{\mathbf{v}}^\top Q_{\mathbf{v}} = Q_{\mathbf{v}}^2 = I - 4\mathbf{v}\mathbf{v}^\top + 4\mathbf{v}\mathbf{v}^\top\mathbf{v}\mathbf{v}^\top = I$$

Property 3 follows since

$$Q_{\mathbf{v}}\mathbf{v} = -\mathbf{v}$$

Property 4 follows since $\mathbf{v}\mathbf{v}^\top$ is a rank-1 matrix as all rows are linear combinations of each other. To see property 5, note there is a dimension $n-1$ space W orthogonal to $\mathbf{v}$, that is, for all $\mathbf{w} \in W$ we have $\mathbf{w}^\top\mathbf{v} = 0$, which implies that

$$Q_{\mathbf{v}}\mathbf{w} = \mathbf{w}$$

In other words, 1 is an eigenvalue with multiplicity $n-1$ and -1 is an eigenvalue with multiplicity 1, and thus the product of the eigenvalues is -1.

∎

Example (reflection through 2-vector) Consider reflection through $\mathbf{x} = [1,2]^\top$. We first need to normalise $\mathbf{x}$:

$$\mathbf{v} = \frac{\mathbf{x}}{\|\mathbf{x}\|} = \begin{bmatrix} \frac{1}{\sqrt{5}} \\ \frac{2}{\sqrt{5}} \end{bmatrix}$$

Note this indeed has unit norm:

$$\|\mathbf{v}\|^2 = \frac{1}{5} + \frac{4}{5} = 1.$$

Thus the reflection matrix is:

$$Q_{\mathbf{v}} = I - 2\mathbf{v}\mathbf{v}^\top = \begin{bmatrix} 1 & \\ & 1 \end{bmatrix} - \frac{2}{5}\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix} = \frac{1}{5}\begin{bmatrix} 3 & -4 \\ -4 & -3 \end{bmatrix}$$

Indeed it is symmetric, and orthogonal. It sends $\mathbf{x}$ to $-\mathbf{x}$:

$$Q_{\mathbf{v}}\mathbf{x} = \frac{1}{5}\begin{bmatrix} 3-8 \\ -4-6 \end{bmatrix} = -\mathbf{x}$$

Any vector orthogonal to $\mathbf{x}$, like $\mathbf{y} = [-2,1]^\top$, is left fixed:

$$Q_{\mathbf{v}}\mathbf{y} = \frac{1}{5}\begin{bmatrix} -6-4 \\ 8-3 \end{bmatrix} = \mathbf{y}$$

Note that *building* the matrix `Q_v` will be expensive ($O(n^2)$ operations), but we can apply $Q_{\mathbf{v}}$ to a vector in $O(n)$ operations using the expression:

$$Q_{\mathbf{v}}\mathbf{x} = \mathbf{x} - 2\mathbf{v}(\mathbf{v}^\top\mathbf{x}).$$

Just as rotations can be used to rotate vectors to be aligned with coordinate axis, so can reflections, but in this case it works for vectors in $\mathbb{R}^n$, not just $\mathbb{R}^2$:

Lemma (Householder reflection) Define $\mathbf{y} = \pm\|\mathbf{x}\|\mathbf{e}_1 + \mathbf{x}$ and $\mathbf{w} = \frac{\mathbf{y}}{\|\mathbf{y}\|}$. Then

$$Q_{\mathbf{w}}\mathbf{x} = \|\mathbf{x}\|\mathbf{e}_1$$

Proof Note that

$$\|\mathbf{y}\|^2 = 2\|\mathbf{x}\|^2 \pm 2\|\mathbf{x}\|x_1,$$
$$\mathbf{y}^\top\mathbf{x} = \|\mathbf{x}\|^2 \pm \|\mathbf{x}\|x_1$$

where $x_1 = \mathbf{e}_1^\top\mathbf{x}$. Therefore:

$$Q_{\mathbf{w}}\mathbf{x} = (I - 2\mathbf{w}\mathbf{w}^\top)\mathbf{x} = \mathbf{x} - 2\frac{\mathbf{y}\|\mathbf{x}\|}{\|\mathbf{y}\|^2}(\|\mathbf{x}\| \pm x_1) = \mathbf{x} - \mathbf{y} = \mp\|\mathbf{x}\|\mathbf{e}_1.$$

∎