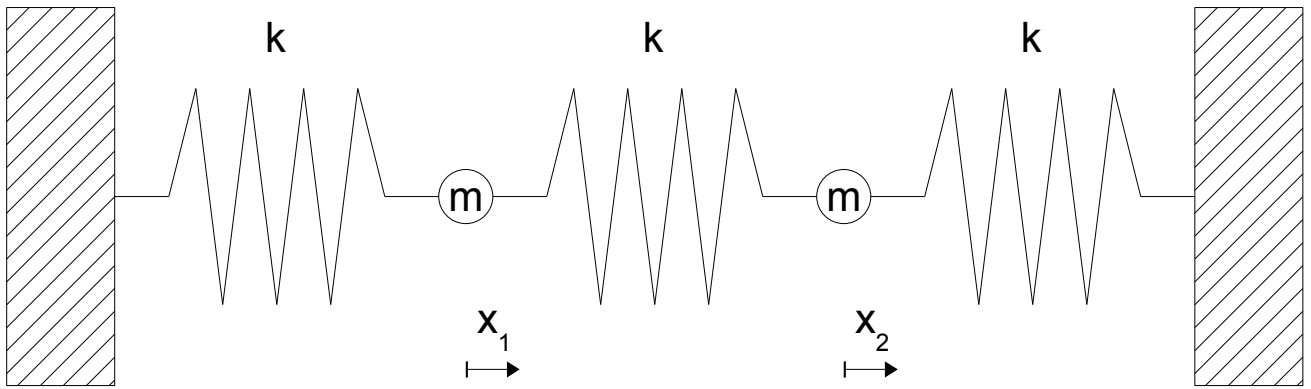
7. The Symmetric Eigenvalue Problem

A ‘must’ for engineering applications . . .

7.1. Motivation

7.1.1. An Example from Physics

Consider the following system, consisting of two bodies B_1, B_2 with mass m, connected by springs with spring constant k. Let $x_1(t), x_2(t)$ denote the displacement of the bodies B_1, B_2 at time t.



From Newton's law ($F = m\ddot{x}$) and Hooke's law ($F = mk$), we have the following **linear second-order differential equations with constant coefficients**:

$$\begin{aligned} m\ddot{x}_1 &= -kx_1 + k(x_2 - x_1) \\ m\ddot{x}_2 &= -k(x_2 - x_1) - kx_2 \end{aligned}$$

- From calculus, we know that we can use the following complex ansatz to find a (non-trivial) solution of this differential equation:

$$x_j(t) = c_j e^{i\omega t},$$

$c_j, \omega \in \mathbb{R}$, $j = 1, 2$.

- Since $\ddot{x}_j(t) = -c_j \omega^2 e^{i\omega t}$, we have:

$$\begin{aligned} -mc_1\omega^2 &= -kc_1 + k(c_2 - c_1) = k(-2c_1 + c_2) \\ -mc_2\omega^2 &= -k(c_2 - c_1) - kc_2 = k(c_1 - 2c_2) \end{aligned}$$

- We substitute $\lambda := -m\omega^2/k$:

$$\begin{aligned} \lambda c_1 &= -2c_1 + c_2 \\ \lambda c_2 &= c_1 - 2c_2 \end{aligned}$$

Or, with $c := (c_1, c_2)^T$,

$$\begin{pmatrix} -2 & 1 \\ 1 & -2 \end{pmatrix} c = \lambda c$$

We have an instance of the **symmetric eigenvalue problem**!

- For given **initial values**, the solution is unique.
- We can obtain similar instances of the symmetric eigenvalue problem for systems with a higher number of bodies.

7.1.2. Eigenvalues in Numerics

- A common problem in numerics is to solve systems of linear equations of the form $Ax = b$ with $A \in \mathbb{R}^{n \times n}$ symmetric, $b \in \mathbb{R}^n$.
- An important question is the **condition** of this problem, i.e. to which extent an error in b affects the value of the correct solution x^*.
- The maximum ratio of the relative error in the solution x^* to the relative error in b (measured using the Euclidean norm) is the **condition number** $\kappa(A)$ of solving $Ax = b$. It can be shown that, for symmetric A,

$$\kappa(A) = \left| \frac{\lambda_{max}(A)}{\lambda_{min}(A)} \right|.$$

  That means the condition depends directly on the eigenvalues of A!
- This number plays an important role in iterative solution of linear equation systems, for example:
  - Assume we have an approximate solution $\hat{x}$.
  - This means we have exactly solved the system $Ax = \hat{b}$ with $\hat{b} := A\hat{x}$.
  - However, if $\kappa(A)$ is large, this means that even if $\hat{b} - b$ is small, our solution may be far away from the solution of $Ax = b$!
- Finally, also the **convergence speed** of common iterative solvers is greatly affected by the eigenvalues of A.

7.2. Condition

- Before trying to develop numerical algorithms for the symmetric eigenvalue problem, we should have a look at its **condition**!
- Assume that instead of A we have a disturbed matrix $A + \varepsilon B$, where $||B||_2 = 1$. Since A is symmetric, we assume that B is also symmetric (usually only one half of A is stored in memory).
- Let λ be an eigenvalue of A and x an eigenvector to this eigenvalue, i.e. $Ax = \lambda x$. Let $x(\varepsilon)$ and $\lambda(\varepsilon)$ be the disturbed values of x and λ, implicitly given by the equality:

$$(A + \varepsilon B)x(\varepsilon) = \lambda(\varepsilon)x(\varepsilon)$$

- Using Taylor series expansion around $\varepsilon_0 = 0$, we have:

$$\begin{aligned}(A + \varepsilon B)(x + x'(0)\varepsilon + \tfrac{1}{2}x''(0)\varepsilon^2 + \dots) = \quad & (\lambda + \lambda'(0)\varepsilon + \tfrac{1}{2}\lambda''(0)\varepsilon^2 + \dots) \\ & (x + x'(0)\varepsilon + \tfrac{1}{2}x''(0)\varepsilon^2 + \dots)\end{aligned}$$

- We are interested in the value of $\lambda'(0)$. Comparing coefficients of ε, we have:

$$\begin{aligned} Bx + Ax'(0) &= \lambda'(0)x + \lambda x'(0) \\ x^T Bx + x^T Ax'(0) &= \lambda'(0)x^T x + \lambda x^T x'(0) \\ x^T Bx + \lambda x^T x'(0) &= \lambda'(0)x^T x + \lambda x^T x'(0) \\ \frac{x^T Bx}{x^T x} &= \lambda'(0) \end{aligned}$$

- It follows that

$$\lambda(\varepsilon) - \lambda(0) = \frac{x^T B x}{x^T x}\varepsilon + O(\varepsilon^2).$$

- Since $\left|\frac{x^T B x}{x^T x}\right| \leq \|B\|_2$, we get

$$|\lambda(\varepsilon) - \lambda(0)| \leq |\varepsilon| \, \|B\|_2 + O(\varepsilon^2)$$

- Finally, with our assumption $\|B\|_2 \leq 1$ for the perturbation matrix B, we have

$$|\lambda(\varepsilon) - \lambda(0)| = O(\varepsilon)$$

- Thus, the **symmetric** eigenvalue problem is a **well-conditioned** problem!

Remark: The **asymmetric** eigenvalue problem is **ill-conditioned**! Consider the asymmetric matrices

$$A = \begin{pmatrix} 1 & 1 \\ 0 & 1 + 10^{-10} \end{pmatrix}, \quad A + \varepsilon B = \begin{pmatrix} 1 & 1 \\ 10^{-10} & 1 + 10^{-10} \end{pmatrix}.$$

A has eigenvalues $\lambda_1 = 1, \lambda_2 = 1 + 10^{-10}$ while $A + \varepsilon B$ has eigenvalues $\mu_{1/2} = 1 \pm 10^{-5}$. This means the error in the eigenvalues is about 10^5 times larger than the error in the matrix!

Naive Computation

- A naive method to compute eigenvalues could be to determine the characteristic polynomial $p(\lambda)$ of A, then using an iterative method for solving $p(\lambda) = 0$, e.g. Newton's iteration.
- However, this reduces the well-conditioned symmetric eigenvalue problem to the **ill-conditioned** problem of determining the roots of a polynomial!
- Example: Consider a 12×12 matrix with the eigenvalues $\lambda_i = i$. Its characteristic polynomial is

$$p(\lambda) = \prod_{i=1}^{12} (\lambda - i)$$

  The coefficient of λ^7 is -6926634. Assume we have the polynomial $q(\lambda) = p(\lambda) - 0.001\lambda^7$, i.e. the relative error of the coefficient of λ^7 is $\varepsilon \approx 1.44 \cdot 10^{-10}$. However, the relative error of the eigenvalue λ_9 in this case is ≈ 0.02!
- We ought to have some better ideas ...

7.3. Vector Iteration

7.3.1. Power Iteration

Let $A \in \mathbb{R}^{n\times n}$ be symmetric. Then A has n eigenvectors $u_1, \ldots, u_n$ that form a basis of $\mathbb{R}^n$, and we can write any vector $x \in \mathbb{R}^n$ as a linear combination of $u_1, \ldots, u_n$:

$$x = c_1u_1 + c_2u_2 + \cdots + c_nu_n \quad (c_1, \ldots, c_n \in \mathbb{R})$$

Let $\lambda_1, \ldots, \lambda_n$ be the eigenvalues of A corresponding to the eigenvectors $u_1, \ldots, u_n$. Then

$$\begin{aligned}
Ax &= \lambda_1c_1u_1 + \lambda_2c_2u_2 + \cdots + \lambda_nc_nu_n \\
A^2x &= \lambda_1^2c_1u_1 + \lambda_2^2c_2u_2 + \cdots + \lambda_n^2c_nu_n \\
&\vdots \\
A^kx &= \lambda_1^kc_1u_1 + \lambda_2^kc_2u_2 + \cdots + \lambda_n^kc_nu_n \\
&= \lambda_1^k\left[c_1u_1 + \left(\frac{\lambda_2}{\lambda_1}\right)^k c_2u_2 + \cdots + \left(\frac{\lambda_n}{\lambda_1}\right)^k c_nu_n\right]
\end{aligned}$$

Assume that λ_1 is **dominant** (i.e. $|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|$).

Then for $k \to \infty$, we have $\left(\frac{\lambda_2}{\lambda_1}\right)^k, \ldots, \left(\frac{\lambda_n}{\lambda_1}\right)^k \to 0$, and thus $\frac{1}{\lambda_1^k}A^kx \to c_1u_1$.

- By choosing an appropriate start vector $x^{(0)} \in \mathbb{R}^n$ and performing the iteration

$$x^{(k+1)} = Ax^{(k)},$$

we can calculate an approximation of an eigenvector of A belonging to the eigenvalue λ_1.

- "Appropriate" means that $u_1^T x^{(0)} \neq 0$. (Otherwise, $x^{(k)}$ will converge to the first eigenvector u_i where $u_i^T x^{(0)} \neq 0$.) If $x^{(0)}$ is chosen randomly with uniform probability, then the probability that $u_1^T x^{(0)} = 0$ is negligible (mathematically, it is zero).
- The values of the iteration vector get arbitrarily large (if $\lambda_1 > 1$) or small (if $\lambda_1 < 1$). To avoid numerical problems, we should therefore normalize the value of x_k after each iteration step.
- Calculating eigenvalues from eigenvectors: Let x be an eigenvector of A belonging to the eigenvalue λ. Then

$$\begin{aligned} Ax &= \lambda x \\ \frac{x^T Ax}{x^T x} &= \lambda \end{aligned}$$

If x is normalized, i.e. $\|x\| = 1$, then $\lambda = x^T Ax$.

- The term

$$\frac{x^T Ax}{x^T x}$$

is also called **Rayleigh quotient**.

Power Iteration: Algorithm

- Choose a random $x^{(0)} \in \mathbb{R}^n$ such that $\left\|x^{(0)}\right\| = 1$
- for $k = 0, 1, 2, \cdots$:
  1. $w^{(k)} := Ax^{(k)}$
  2. $\lambda^{(k)} := (x^{(k)})^T w^{(k)}$
  3. $x^{(k+1)} := \frac{w^{(k)}}{\left\|w^{(k)}\right\|}$
  4. stop if approximation "sufficiently accurate"

Remarks:

- We can consider our approximation $x^{(k)}$ "sufficiently accurate" if

$$\left\|w^{(k)} - \lambda^{(k)} x^{(k)}\right\| \leq \varepsilon \left\|w^{(k)}\right\|.$$

- The **cost** per iteration step is determined by the cost of computing the matrix-vector product $Ax^{(k)}$, which is at most $O(n^2)$.

Convergence

- The **convergence order** of the sequence $\{x^{(k)}\}$ is **linear** with convergence rate $q = \lambda_2/\lambda_1$.
- The convergence order of the sequence $\{\lambda^{(k)}\}$ is **quadratic** with convergence rate q.
- Thus, if λ_2 is only slightly smaller than λ_1, convergence will be very slow. We can address this problem by **shifting** the eigenvalues:
  - Assume we have guessed an approximation $\mu \approx \lambda_2$.
  - Consider the matrix $A - \mu I$. It is easy to see that this matrix has eigenvalues $\lambda_1 - \mu, \ldots, \lambda_n - \mu$.
  - By performing the iteration with the matrix $A' = A - \mu I$ instead of A, we can greatly speed up convergence.
  - Example: $n = 2, \lambda_1 = 5, \lambda_2 = 4.9$.
    Using $\mu = 4.85$, we have convergence rate $\frac{\lambda_2 - \mu}{\lambda_1 - \mu} = \frac{1}{3}$.
    Compare this to the "original" convergence rate $\lambda_2/\lambda_1 = \frac{49}{50}$!
  - In general, we have a convergence rate $\frac{\lambda_j - \mu}{\lambda_i - \mu}$, where $\lambda_i - \mu$ is the largest and $\lambda_j - \mu$ the second largest shifted eigenvalue, regarding the modulus, i.e. $|\lambda_i - \mu| \geq |\lambda_j - \mu| \geq |\lambda_k - \mu| \ \ \forall\, k \in \{1, \ldots, n\} \setminus \{i, j\}$.

Remarks

- The algorithm also works for a slightly weaker condition on the eigenvalues of A: If A has no dominant eigenvalue, but there exists an $r \in \{1, \ldots, n\}$ such that $\lambda_1 = \cdots = \lambda_r$ and $|\lambda_r| > |\lambda_i|$ for all $i \in \{r+1, \ldots, n\}$, then the sequence $\{x^{(k)}\}$ converges to a linear combination of $u_1, \ldots, u_r$.
- Power iteration can be used to successively compute several eigenvalues in descending order:
  - It is a standard result in linear algebra that, if v_1 is an eigenvector belonging to the eigenvalue λ_1, the matrix $A - \lambda_1 v_1 v_1^T$ has the same eigenvalues and eigenvectors as A, but λ_1 has been replaced by 0.
  - After eigenvalue λ_1 and a corresponding eigenvector v_1 have been computed with sufficient precision, we continue the iteration with the matrix $A - \lambda_1 v_1 v_1^T$ to find λ_2 and so on.
- Prominent areas of application of power iteration are found in statistics (Google's PageRank algorithm, Principal Component Analysis).

7.3.2. Inverse Iteration

- We now look for a method to compute a **specific** eigenvalue λ^* of a symmetric matrix $A \in \mathbb{R}^{n \times n}$, given the approximation $\mu \approx \lambda^*$. Let $\lambda_1, \ldots, \lambda_n$ be the eigenvalues of A.
- Remember that A^{-1} has eigenvalues $\lambda_1^{-1}, \ldots, \lambda_n^{-1}$, such that we could compute the **smallest** eigenvalue of A by perfoming power iteration with A^{-1}.
- We can combine this with the shifting technique we saw before: If μ is closer to λ^* than to any other eigenvalue, $\lambda^* - \mu$ is the smallest eigenvalue of $A - \mu I$.
- We can therefore perform power iteration with $(A - \mu I)^{-1}$ to calculate an approximation for $\lambda^* - \mu$!
- However, we do not have to calculate $(A - \mu I)^{-1}$ explicitly since the recurrence

$$x^{(k+1)} = (A - \mu I)^{-1} x^{(k)}$$

is equivalent to

$$(A - \mu I) x^{(k+1)} = x^{(k)}.$$

This means that we can calculate the value of $x^{(k+1)}$ by using a linear system solver in each iteration step.

Inverse Iteration: Algorithm

- Choose a random $x^{(0)} \in \mathbb{R}^n$ such that $\left\|x^{(0)}\right\| = 1$
- for $k = 0, 1, 2, \cdots$:
  1. solve $(A - \mu I)w^{(k)} = x^{(k)}$
  2. $x^{(k+1)} := \frac{w^{(k)}}{\left\|w^{(k)}\right\|}$
  3. Stop if approximation “sufficiently accurate”

Remarks:

- The cost of each iteration step is dominated by the complexity of solving the linear system with coefficient matrix $A - \mu I$.
- Since this matrix does not change during iteration, we can compute its LU factorization in the beginning, thereby reducing the cost of each iteration step to $O(n^2)$ operations.
- As we are essentially performing power iteration, the same remarks about convergence apply.

7.3.3. Rayleigh Quotient Iteration

- In our implementation of power iteration, we used the Rayleigh quotient to compute an approximation of λ_1 in each iteration step.
- We can make use of this value to refine our approximation μ in each step!
- This gives the following algorithm:
  - Choose a random $x^{(0)} \in \mathbb{R}^n$ such that $\left\|x^{(0)}\right\| = 1$
  - for $k = 0, 1, 2, \cdots$:
    1. $\mu^{(k)} := (x^{(k)})^T A x^{(k)}$
    2. solve $(A - \mu^{(k)} I) w^{(k)} = x^{(k)}$
    3. $x^{(k+1)} := \frac{w^{(k)}}{\left\|w^{(k)}\right\|}$
    4. Stop if approximation “sufficiently accurate”
- This algorithm is usually known as **Rayleigh quotient iteration**.

Remarks

- It can be shown that the convergence order of the sequence $\{\lambda^{(k)}\}$ is now **cubic**! Still, the sequence of the eigenvectors converges linearly.
- However, since we have to solve a different system of linear equations in each iteration step, the **cost** per step is usually significantly higher than for power iteration!
- Rayleigh quotient iteration is the method of choice if we have a good approximation of the eigenvalue we look for – then only a few iteration steps are needed due to the high convergence order.
- If we already have an eigenvalue and only want to compute a corresponding eigenvector, “standard” inverse iteration has an advantage over Rayleigh quotient iteration, since the sequence of eigenvectors converges linearly for both methods, but Rayleigh quotient iteration is more costly.

7.4. QR Iteration

- **Factorization methods** such as the QR iteration [Francis 1961] compute **all** eigenvalues of a symmetric matrix $A \in \mathbb{R}^{n \times n}$ at once.
- The idea is to compute a sequence of matrices $\{A_k\}$ that are **similar** to A. This sequence should converge to a matrix in triangular form, such that the diagonal of A_k is an approximation for the eigenvalues of A.
- The QR iteration is based on the QR decomposition which factorizes a given matrix A such that $A = QR$, with Q an orthogonal and R an upper triangular matrix. Then the matrix

$$RQ = Q^{-1}AQ = Q^T AQ$$

  is similar to A.
- It can be shown that the sequence $\{A^{(k)}\}$ defined by

$$\begin{aligned} A^{(0)} &= A, \\ A^{(k+1)} &= R^{(k)}Q^{(k)}, \\ & \quad Q^{(k)} \text{ orthogonal, } R^{(k)} \text{ upper triangular, } A^{(k)} = Q^{(k)}R^{(k)} \end{aligned}$$

  converges to a diagonal matrix.

QR Iteration: Algorithm

- Thus, we have the following algorithm:
  - $A^{(0)} := A$
  - for $k = 0, 1, 2, \ldots$:
    1. Compute a factorization $A^{(k)} = QR$ using QR decomposition
    2. $A^{(k+1)} := RQ$
    3. if $a_{n,n-1}^{(k+1)}$ sufficiently small:
       output $\lambda_n \approx a_{n,n}^{(k+1)}$
       remove row and column n
- We consider the subdiagonal element $a_{n,n-1}^{(k+1)}$ "sufficiently small" if its absolute value is $\leq \varepsilon \cdot |a_{n,n}^{(k+1)}|$.
- We still need to know how to implement the QR decomposition!

7.4.1. QR Decomposition

- Remember: We want to decompose a matrix A into an orthogonal matrix Q and an upper triangular matrix R such that $A = QR$.
- This problem is equivalent to the following: Find an orthogonal matrix G such that $GA = R$, i.e. G "zeroes out" the subdiagonal part of A. Then $A = G^T R$.
- We first have a look at the two-dimensional case: Given a vector $x = (x_1, x_2)^T$, find an orthogonal matrix G such that $Gx = (r, 0)^T$ for some $r \in \mathbb{R}$.
- It is easy to see that

$$G = \frac{1}{\sqrt{x_1^2 + x_2^2}} \begin{pmatrix} x_1 & x_2 \\ -x_2 & x_1 \end{pmatrix}$$

  does the job.
- Geometrically, the matrix G describes a rotation by $\theta = \arctan(x_2/x_1)$ radians in the (x_1, x_2) plane.
- G is called a **Givens** rotation matrix [Givens 1958].

- Generalization to the n-dimensional case: When multiplied with A,

$$G_{ij} = \begin{pmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & \rho \cdot a_{jj} & \cdots & \rho \cdot a_{ij} & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -\rho \cdot a_{ij} & \cdots & \rho \cdot a_{jj} & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{pmatrix}, \quad \rho = \frac{1}{\sqrt{a_{ij}^2 + a_{jj}^2}}$$

  affects rows i and j of A and eliminates entry a_{ij}. G_{ij} differs from the identity matrix in only four entries.

- To zero out the entire subdiagonal part, we use the product

$$G = G_{n-1,n} \cdot G_{n-2,n} G_{n-2,n-1} \cdot \ldots \cdot G_{2,n} \cdots G_{2,3} \cdot G_{1,n} \cdots G_{1,2},$$

  i.e. we eliminate the matrix entries column-wise from left to right, top to bottom, in order not to introduce additional non-zero entries!

- Since the product of two orthogonal matrices is orthogonal, G is orthogonal and GA is an upper triangular matrix for any matrix A.

QR Iteration: Implementation

- We now have a method to compute the QR decomposition of A: $R = GA$, $Q = G^T$.
- The naive computation of G via $n(n-1)/2$ matrix multiplications costs $O(n^5)$ operations!
- However, the **effect** of the multiplication with a Givens rotation matrix can be computed with $O(n)$ operations due to its special structure. This leads to a total of $O(n^3)$ operations for the QR decomposition.
- For QR iteration, we are not even interested in the value of G. We only need to compute the product $A^{(k+1)} = GA^{(k)}G^T$!
- Since matrix multiplication is associative and $(M_1 M_2)^T = M_2^T M_1^T$, we can use the following scheme for computation of $A^{(k+1)}$:

$$A^{(k+1)} = \ldots G_{1,3} \left(G_{1,2} \, A^{(k)} \, G_{1,2}^T \right) G_{1,3}^T \ldots$$

Remarks

- QR iteration is the method of choice if **all** eigenvalues of a matrix are needed.
- The convergence order of the sequence $\{A^{(k)}\}$ is **linear**.
- As with power iteration, convergence is slow if any of the eigenvalues are close together. By using a shift operation, similar to the one in Rayleigh quotient iteration, we can achieve **quadratic** convergence order.
- Our algorithm does not calculate eigenvectors. Eigenvectors to specific eigenvalues can be retrieved using inverse iteration.
- QR decomposition can also be used as an **exact solver** for systems of linear equations: Let $A \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^n$. Then

$$\begin{aligned} & Ax &=& b \\ \iff & QRx &=& b \\ \iff & Rx &=& Q^T b \end{aligned}$$

  - Since R is upper triangular, the system $Rx = Q^T b$ can be solved with $O(n^2)$ operations.
  - As the computation of the QR decomposition can take $O(n^3)$ operations, this method – as LU factorization – is especially useful when a set of systems with the same matrix A but many different vectors b_i has to be solved.

7.5. Reduction Algorithms

- The cost of all three algorithms presented so far depends largely on the number of non-zero entries in the matrix A:
  - Cost of an iteration step in power iteration is dominated by the complexity of the matrix-vector product Ax.
  - Cost of an iteration step in inverse iteration is dominated by the complexity of solving $Ax = b$.
  - Cost of an iteration step in QR iteration depends on the number of Givens rotations that we have to use to eliminate non-zero entries in the iteration matrix A_k.
- Therefore, for numerical computation of eigenvalues, we are interested in matrices that are **similar** to A but have a significantly lower number of non-zero entries.

- We try to develop an algorithm that transforms a symmetric matrix $A \in \mathbb{R}^{n \times n}$ into a similar **tridiagonal** symmetric matrix:

$$A \to \begin{pmatrix} * & * & 0 & 0 & \cdots & \cdots & 0 \\ * & * & * & 0 & 0 & \cdots & 0 \\ 0 & * & * & * & 0 & \cdots & 0 \\ & & & \vdots & & & \\ 0 & \cdots & 0 & * & * & * & 0 \\ 0 & \cdots & 0 & 0 & * & * & * \\ 0 & \cdots & \cdots & 0 & 0 & * & * \end{pmatrix}$$

- Observation: Matrix $A \in \mathbb{R}^{n \times n}$ is tridiagonal and symmetric if and only if it is of the form

$$\begin{pmatrix} \alpha & \rho & 0 & \cdots & 0 \\ \rho & & & & \\ 0 & & A' & & \\ \vdots & & & & \\ 0 & & & & \end{pmatrix}$$

  with $A' \in \mathbb{R}^{(n-1)\times(n-1)}$ tridiagonal and symmetric.

- This leads to a “dynamic programming” approach: Use a similarity transformation to transform A to a matrix in the above form, then continue with A' recursively.

Transformation Algorithm

- If we had a matrix $H' \in \mathbb{R}^{(n-1)\times(n-1)}$ such that

$$H' \begin{pmatrix} a_{12} \\ a_{13} \\ \vdots \\ a_{1n} \end{pmatrix} = \begin{pmatrix} \rho \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

for some $\rho \in \mathbb{R}$, then

$$HA := \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & & & \\ 0 & & H' & \\ \vdots & & & \\ 0 & & & \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{12} & & & \\ a_{13} & & & \\ \vdots & & \ddots & \vdots \\ a_{1n} & & \cdots & a_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \rho & & & \\ 0 & & B & \\ \vdots & & & \\ 0 & & & \end{pmatrix}$$

($B \in \mathbb{R}^{(n-1)\times(n-1)}$).

- We already know that such a H' exists – for example, it can be described as the product of $n-2$ Givens rotations.

- By transposing the result, we can use the effect of multiplication with H again:

$$H(HA)^T = H \begin{pmatrix} a_{11} & \rho & 0 & \cdots & 0 \\ a_{12} & & & & \\ a_{13} & & B^T & & \\ \vdots & & & & \\ a_{1n} & & & & \end{pmatrix} = \begin{pmatrix} a_{11} & \rho & 0 & \cdots & 0 \\ \rho & & & & \\ 0 & & A_2 & & \\ \vdots & & & & \\ 0 & & & & \end{pmatrix}$$

- Since $H(HA)^T = HA^T H^T = HAH^T$, this is a similarity transformation if and only if H is orthogonal.
- For that, it is important that the application of H does not destroy the first *row* of A. Therefore, we can only eliminate everything below the subdiagonal element of the first column, and produce a triadiagonal matrix only (instead of a diagonal one).
- With basic linear algebra, one can also show that A_2 is symmetric.
- Our task is now to find an appropriate matrix H'!

Householder Transformations

- A first idea could be to choose H' as a product of Givens rotations, as previously indicated.
- However, in practice **Householder transformations** [Householder 1958] are used more frequently.
- For any vector $x \in \mathbb{R}^n$, **Householder matrix** $H \in \mathbb{R}^{n \times n}$ is defined by

$$H = I - 2vv^T,$$

with $v := \frac{u}{\|u\|_2}$ and $u := x - \|x\|_2 e_1$, such that

$$Hx = \begin{pmatrix} \rho \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

for some $\rho \in \mathbb{R}$.

- Geometrically, H describes a **reflection** of x in the hyperplane defined by the vector v which is orthogonal to the hyperplane.

Remarks

- As with Givens rotation matrices, Householder matrices are usually not explicitly computed. When using an appropriate implementation, the effect of multiplication with a Householder matrix can be computed with $O(n^2)$ operations.
- In that case, the transformation algorithm described above takes $O(n^3)$ steps, rather than $O(n^4)$ steps for a naive implementation with $O(n)$ matrix multiplications.
- Householder transformations can also be used to compute the QR decomposition, since the product of $n-1$ Householder matrices transforms a matrix into upper triangular form.
- We could therefore also use them for QR iteration!
- However, it is more efficient to first reduce a matrix to tridiagonal form using Householder transformations (cost: $O(n^3)$), then perform QR iteration using Givens rotations on the reduced matrix.
- Then, in each iteration step, only $O(n)$ elements need to be zeroed using Givens rotations, which results in a cost of $O(n^2)$ per iteration step, vs. $O(n^3)$ when using Householder transformations.