10-725/36-725: Convex Optimization **Fall 2015**

Lecture 19: November 5

Lecturer: Ryan Tibshirani *Scribes: Bohan Li, Donghan Yu, Ge Huang*

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

19.1 Flops for basic operations

Complexity can be expressed in terms of *floating point operations* or *flops* required to find the solution. A flop serves as a basic unit of computation, which could denote one addition, subtraction, multiplication or division of floating point numbers. Note that, the flop count is just a rough measure of how expensive an algorithm can be. Many more aspects need to be taken into account to accurately estimate practical runtime. And in practical situations, we're interested in rough, not exact flop counts to measure the complexity of operations.

In the following sections, we'll show the flop count of some basic operations.

19.1.1 Vector-vector opertaions

Given vector $a, b \in \mathbb{R}^n$:

- Addition $a + b$: requires n flops for n element-wise additions.
- Scalar multiplication $c \cdot a$: requires n flops for n element-wise multiplications.
- Inner product $a^T b$: requires approximately $2n$ flops for n multiplications and $n - 1$ additions.

However, as said above, the flop count is just a rough measure of how expensive an algorithm can be. For example, setting every element of vector a to 1 costs 0 flops.

19.1.2 Matrix-vector opertaions

Given $A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^n$, consider Ab:

- In general, $Ab = (a_1^T, a_2^T, \cdots, a_m^T)^T b = (a_1^T b, a_2^T b, \cdots, a_m^T b)^T$, each row takes $2n$ flops, then m rows take $2mn$ flops in total.
- If A is s-sparse, then the i'th element of Ab is $Ab(i) = \sum_j a_{ij} b_j,\ (i,j) \in S$, where S is the index set of non-zero elements in A. Since $|S| = s$, the total flop count is $2s$. (The worst case is that all the non-zero elements are in the same row)

- If $A \in \mathbb{R}^{n\times n}$ is k-banded, the non-zero elements of each row is $2k$, then the total flop count of n row is $2nk$.
- If $A = \sum_{i=1}^{r} u_i v_i^T \in R^{m\times n}$, $Ab = \sum_{i=1}^{r} u_i(v_i^T b)$. Calculate $m_i = v_i^T b,\ i = 1, \cdots, r$ costs $2nr$ flops. Then scalar multiplication takes mr flops, finally the summation takes mr flops. The total flop count is $2r(m+n)$.
- If $A \in \mathbb{R}^{n\times n}$ is a permutation matrix, it takes 0 flops to reorder elements in b.

19.1.3 Matrix-matrix product

Given $A \in \mathbb{R}^{m\times n}, B \in \mathbb{R}^{n\times p}$, consider AB:

- In general, $AB = A(b_1, b_2, \cdots, b_p) = (Ab_1, \cdots, Ab_p)$. For each b_i, the product cost $2mn$ flops. Then the total flop count is $2mnp$.
- If A is s-sparse, it costs $2sp$ flops. The cost can be further reduced if B is also sparse.

19.1.4 Matrix-matrix-vector product

Given $A \in \mathbb{R}^{m\times n}, B \in \mathbb{R}^{n\times p}, c \in \mathbb{R}^p$, consider ABc:

- If product is done properly, that is, $ABc = A(Bc)$, the total cost is $2np+2mn$. Else if done improperly, i.e., $ABc = (AB)c$, the cost is $2mnp + 2mp$!

19.2 Solving linear systems

Given a non-singular square matrix $A \in \mathbb{R}^{n\times n}$ and a vector $b \in \mathbb{R}^n$, consider solving the linear equation, $Ax = b$. In others words, we intend to determine the cost of computing $x = A^{-1}b$. Note that in Newton's method, we need to solve $\nabla^2 f(x)v = -\nabla f(x)$, which is exactly this form.

- In general, it cost n^3 flops. This can be a very expensive cost when n is a large number. However, the complexity of solving linear systems can be reduced for some matrices having special properties.
- If A is diagnal, it just costs n flops, one each for element-wise divisions. $x = (b_1/a_1, \cdots, b_n/a_n)$.
- If A is lower triangular ($A_{ij} = 0, j > i$), it costs about n^2 flops by forward substitution.

$$\begin{aligned}
&x_1 = b_1/A_{11}\\
&x_2 = (b_2 - A_{21}x_1)/A_{22}\\
&\cdots\\
&x_n = (b_n - A_{n,n-1}x_{n-1} - \cdots - A_{n,1}x_1)/A_{nn}
\end{aligned}$$

- If A is upper triangular ($A_{ij} = 0, j > i$), it costs about n^2 flops by back substitution.
- If A is s-sparse, it often costs $\ll n^3$. However, it is hard to determine the exact order of flops. It heavily depends on the sparsity structure of the matrix.

- If A is k-banded, it costs about nk^2 flops.
- If A is a permutation matrix, which means that $A^{-1} = A^T$. Then $x = A^T b$ costs 0 flops since each row of A has only one element and x can be obtained from n assignment operations that are free of cost.

19.3 Matrix factorizations

To solve a linear system $Ax = b$, instead of doing $A\backslash b$ or compute A^{-1} directly, it is useful to instead factorize A into product of some structured (orthogonal, triangular, diagonal, or permutation matrices that are easier to compute inverse) $(A_k)'s$. Here we are going to introduce another two very useful alternatives, the QR decomposition and Cholesky decomposition.

19.3.1 QR decomposition

QR decomposition works for a more general case even when the matrix under consideration is not square. Any matrix $A \in R^{m\times n}$ can be decomposed into the form as:

$$A = QR$$

where $m \geq n, Q \in R^{m\times n}, Q^T Q = I_n$ (orthogonal) , $R \in R^{n\times n}$ is upper triangular. Facts about the factor matrix Q and R:

- The column vectors of Q = [Q1 , Q2 , , Qn] actually forms the orthonormal basis of a n dimensional subspace of R^m. So it can be treated as orthogonal in a general sense.
- Moreover, if we expand the columns of Q to the whole space as [Q1, Q2, , Qm], then it holds that the column span of Q = [Qr+1, , Qm] actually forms an orthogonal complementary of col(Q). Then by the fact that orthogonal matrix preserves vectors norm, we have

  $$x^T = x^T[Q \ \ \tilde{Q}][Q \ \ \tilde{Q}]^T x = x^T(QQ^T + \tilde{Q}\tilde{Q}^T)x = ||Q^T x||_2^2 + ||\tilde{Q}^T x||_2$$

  which can simply the optimization problem in many cases.
- The diagonal elements of R are relevant to the rank of A. If rank(A) $\geq$ r, then the first r diagonal entries of R are nonzero and span(Q1, , Qr) = col(A) where r $\leq$ n.

Assuming A is nonsingular and square, we can now solve Ax = b:

- Compute $y = Q^T b$ in $2n^2$ flops.
- Solve $Rx = y$, in n^2 flops (back substitution)

So solving costs $3n^2$ flops

19.3.2 Cholesky decomposition

When the matrix A is symmetric and positive definite, i.e $A \in Sn++$, there exists a unique lower triangular matrix L such that $A = LL^T$. Moreover, the matrix L is non-singular. Since Cholesky decomposition is a special case of Gaussian elimination for the positive definite matrices, its computation requires $n^3/3$ flops. To solve a linear equation Ax = b using Cholesky decomposition, the flop number is given by:

- Compute $y = L^1 b$ by forward substitution in n^2 flops.
- Compute $x = (L^T)^{-1} y$ by backward substitution in n^2 flops.

So in general, to solve a n dimensional linear equation by a given Cholesky decomposition only needs $2n^2$ flops.

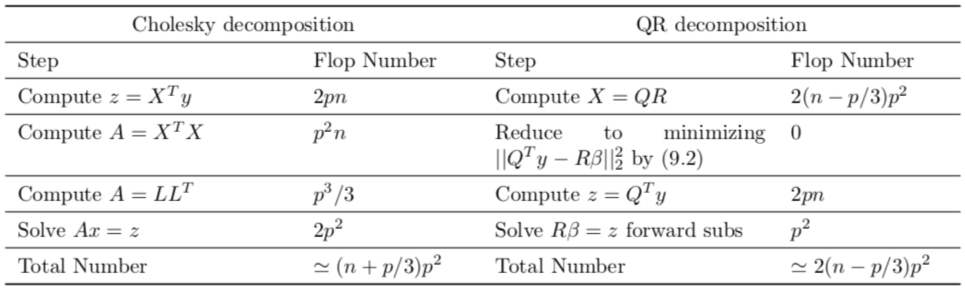
19.3.3 Computational cost of Cholesky and QR on least square

Here we perform an analysis on the computational cost of both sides. The case studied here is least square problem shown below as:

$$min_{\beta \in R^P} ||y - X\beta||_2^2 \rightarrow \beta = (X^T X)^{-1} (X^T y)$$

where $X \in R^{nxp}, y \in R^n$.

To solve this linear equation given by the analytical solution, necessary flop numbers are shown in Table 19.1. This shows that Cholesky decomposition is computationally cheaper than QR decomposition.



Cholesky decomposition		QR decomposition	
Step	Flop Number	Step	Flop Number
Compute $z = X^T y$	$2pn$	Compute $X = QR$	$2(n - p/3)p^2$
Compute $A = X^T X$	$p^2 n$	Reduce to minimizing $\|\|Q^T y - R\beta\|\|_2^2$ by (9.2)	0
Compute $A = LL^T$	$p^3/3$	Compute $z = Q^T y$	$2pn$
Solve $Ax = z$	$2p^2$	Solve $R\beta = z$ forward subs	p^2
Total Number	$\simeq (n + p/3)p^2$	Total Number	$\simeq 2(n - p/3)p^2$

Figure 19.1: When A is in poor condition, gradient descent will spend a lot of time traversing back and forth "across the valley", rather than "down the valley".

19.4 Linear systems and Sensitivity analysis

From the previous section, it seems that Cholesky decomposition is always better than QR decomposition computationally. However, as we take the numerical robustness, the performance of QR will win over by sensitivity analysis. To start with, consider the linear system Ax = b, with nonsingular $A \in R^{n \times n}$. The singular value decomposition of A is A = UVT, where U,V Rnn are orthogonal, and $\Sigma \in R^{n \times n}$ is diagonal with elements $\sigma_1 \geq ... \geq \sigma_n > 0$.

A could be near a singular matrix B even if its full rank, i.e.,

$$dist(A, \mathcal{R}_k) = min_{rand(B)=k} ||A - B||_{op}$$

could be small for some k ¡ n. We can show with SVD analysis that $dist(A, Rk) = \sigma_{k+1}$. If the value is small, solving $x = A^{-1} b$ could be problematic.

Applying SVD we can see that:

$$x = A^{-1} b = V \Sigma^{-1} U^T b = \sum_{i=1}^{n} \frac{v_i u_i^T b}{\sigma_i}$$

If $\sigma_i > 0$ is small, close to set of rank i-1 matrices, that would pose some problem. In precise sensitivity analysis: fix some $F \in R_{nxn}, f \in R_n$, solve:

$$(A + \epsilon F)x(\epsilon) = (b + \epsilon f)$$

Theorem 9.1 The solution to the perturbed system satisfies:

$$\frac{||x(\epsilon) - x||_2^2}{||x||_2} \le \kappa(A)(\rho_A + \rho_b) + O(\epsilon^2)$$

where $\kappa(A) = \frac{\sigma_1}{\sigma_n}$ is the condition number of A, and $\rho_A = |\epsilon|\frac{||F||_{op}}{||A||_{op}}, \rho_b = |\epsilon|\frac{||f||_2}{||b||_2}$ are the relative errors.

Proof: Differentiating the equation above, let $\epsilon = 0$, and solving for $\frac{dx}{d\epsilon}$. We have:

$$\frac{dx}{d\epsilon}(0) = A^{-1}(f - Fx)$$

where $x = x(0)$.

Apply Taylor expansion around 0,

$$x(\epsilon) = x +^{-1}(f - Fx) + O(\epsilon)^2$$

Rearrange and we arrive at the inequality,

$$\frac{||x(\epsilon) - x||_2^2}{||x||_2} \le |\epsilon|||A^{-1}||_{op}(\frac{||f||_2}{||x||_2} + ||F||_{op} + O(\epsilon)^2$$

Multiplying and dividing by $||A||_{Op}$, and note that $\kappa(A) = ||A||_{op}||A_{op}^{-1}$, which proves the result.

In linear systems worse conditioning means great sensitivity.

For least squares problems:$min_{\beta \in R^p}||y - X\beta||_2^2$, Cholesky solves $X^TX\beta = X^Ty$, hence the sensitivity scales with $\kappa(X^TX) = \kappa(X)^2$. While QR operates on X without forming X^TX, that sensitivity scales with $\kappa(X) + \rho_{LS} \times \kappa(X)^2$, where $\rho_{LS} = ||yX^{\top||_2^2}$.

In summary, Cholesky is cheaper and use less memory, while QR is more stable when ρ_{LS} is small and $\kappa(X)$ is large.

19.5 Indirect methods

A counterpart to direct methods is indirect methods, which is more aligned with everything so far that we have learned in this course. These are methods which iteractively produce sequence of estimates $x^{(k)}, k = 1, 2, 3 \ldots$. They will converge to a solution when k goes to infinity. They are most often used for very large and sparse systems.

Actually everything that we have learned so far is iterative or indirect rather than exact. The only exception that we have very briefly talked about is the simplex algorithm. It is a direct non-iterative method for linear programs.

So, when to use direct versus indirect? If you have a matrix that you can fit easily in memory, then you should always use direct methods. In such circumstances, there is no point using indirect methods. One should start considering indirect methods when you have a large enough matrix and just get it into memory is a issue.

19.5.1 Jacobi and Gauss-Seidl

Given $A \in \mathbb{S}^n_{++}$, Jacobi iterations and Gauss-Seidl iterations are the two most basic iterative approaches for solving linear system $Ax = b$. In the next lecture (coordinate descent), we will find out that Jacobi and Gauss-Seidl are exactly the coordinate descent methods for solving the following quadratic minimization problem:

$$\min_x \frac{1}{2}x^T Ax - b^T x$$

Why? As $A \in \mathbb{S}^n_{++}$, note that the function

$$\phi(x) = \frac{1}{2}x^T Ax - b^T x \tag{19.1}$$

is convex, and its minimizer satisfies $0 = \nabla\phi(x) = Ax - b$. Therefore, minimizing ϕ above is equivalent to solving $Ax = b$.

- **Jacobi iterations**. In this method, we initialize $x_0 \in \mathbb{R}^n$, and repeat for $k = 1, 2, 3, \ldots$

$$x_i^{(k)} = \left(b_i - \sum_{j \neq i} A_{ij} x_j^{(k-1)}\right) / A_{ii}, i = 1, \ldots, n \tag{19.2}$$

- **Gauss-Seidl iterations**. In this method, we initialize $x_0 \in \mathbb{R}^n$, and repeat for $k = 1, 2, 3, \ldots$

$$x_i^{(k)} = \left(b_i - \sum_{j<i} A_{ij} x_j^{(k)} - \sum_{j>i} A_{ij} x_j^{(k-1)}\right) / A_{ii}, i = 1, \ldots, n \tag{19.3}$$

Whether using only the most recent iterates is only difference, but it is a huge difference. Actually it makes all the difference because Jacobi iterations generically do not converge; it can only converge over nice situations. On the contrary, Gauss-Seidl iterations always converge. And that is kind of bad news for parallel computation: Gauss-Seidl cannot be parallel, meanwhile Jacobi can.

19.5.2 Gradient descent

How about gradient descent? So let's apply it to this problem: initialize $x^{(0)}$, repeat:

$$x^{(k)} = x^{(k-1)} + t_k r^{(k-1)} \text{ , where } r^{(k-1)} = b - Ax^{(k-1)} \tag{19.4}$$

for $k = 1, 2, 3, \ldots$. Here $r^{(k-1)}$ is the negative gradient.

What step sizes to use? In fact, this is an interesting case where we can do exact step size optimization. We can actually plug in the negative gradient r, and try to find a t which can minimize the $\phi(x + tr)$. In the previous lectures, we emphasized that exact step size optimization was never a good idea unless we had quadratics. Now we are in this special case.

Here we omit the superscripts of $(k - 1)$ for brevity, then the best step size should be given by

$$t_k = \arg\min_{t \leq 0} \phi(x + tr) = \frac{r^T r}{r^T Ar} \tag{19.5}$$

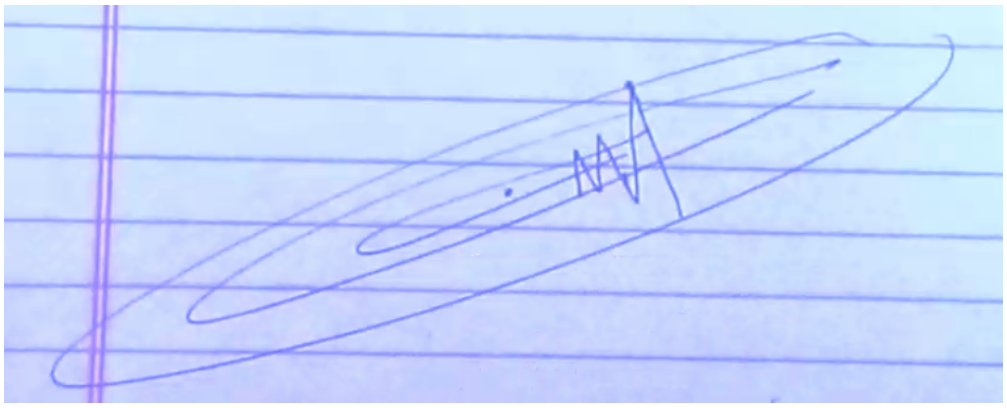


Figure 19.2: When A is in poor condition, gradient descent will spend a lot of time traversing back and forth "across the valley", rather than "down the valley".

Convergence Analysis. Well, what can we say about it? As the quadratic objective ϕ is strongly convex, gradient descent should enjoy **linear convergence**. It should converge quickly like $O(\log(1/\epsilon))$.

For this specific problem (solving linear system with gradient descent), we can say much more precise result in terms of the contraction factor.

From one iteration to the next one, if we take a look at the A-norm distance $x^{(k)}$ and the solution x, it is the decrease from the previous iteration by the amount of $\sqrt{1-\kappa(A)^{-1}}$. We can see that the poorer the condition of A, the slower the convergence is.

Formally, we we have the following theorem.

Theorem 19.1 *Gradient descent with exact step sizes satisfies*

$$\|x^{(k)} - x\|_A \leq \sqrt{1-\kappa(A)^{-1}}\|x^{(k-1)} - x\|_A \tag{19.6}$$

where $\|x\|_A^2 = x^TAx$ *and* $\kappa(A) = \lambda_1(A)/\lambda_n(A)$ *is the condition number of* A.

Proof: The proof is similar to what we did on strong convexity analysis, which was our previous homework. This proof is actually much easier; it is just direct calculation. ■

And an important note is that the contraction factor here depends **adversely** on $\kappa(A)$. To get $\|x^{(k)} - x\|_A \leq \epsilon\|x^{(0))} - x\|_A$, we require $O(\kappa(A)\log(1/\epsilon))$ iterations.

19.5.3 Conjugate gradient

So what is the problem of gradient descent? When $\kappa(A)$ is large, the contour of this function ϕ are elongated ellipsoids. Roughly put, gradient descent will spend a lot of time traversing back and forth "across the valley", rather than "down the valley", as is illustrated in Fig.19.2. In other words, there is **not enough diversity** in the descent directions $r^{(k-1)} = b - Ax^{(k-1)}$.

Conjugate gradient is an extremely clever idea. The purpose of it is to do something like gradient descent, but use a difference direction p that is constructed to be diverse. That is, to repeat iteration like this one:

$$x^{(k)} = x^{(k-1)} + t_k p^{(k-1)} \tag{19.7}$$

and $p^k \in \text{span}\left\{Ap^{(1)}, \dots, Ap^{(k-1)}\right\}^{\perp}$. That is, we expect the new direction to be orthogonal to the past directions after they multiplied by the matrix A. Each pair of p's satisfy A-conjugate with each other. We say p, q are A-conjugate provided $p^T Aq = 0$. This explains the name "conjuage gradient".

Intuition. When we fix the direction p, we are going to do step size optimization. The optimal step size is given by,

$$t_k = \arg\min_{t \le 0} \phi(x + tp) = \frac{p^T r}{p^T Ap} \tag{19.8}$$

Then we plug t_k and p into the obejective function ϕ itself, and get

$$\phi(x^{(k)}) = \phi(x^{(k-1)}) - \frac{1}{2} \frac{(p^{(k)})^T r^{(k-1)}}{(p^{(k)})^T Ap^{(k)}} \tag{19.9}$$

Now we can observe two conflicting goals in conjugate gradient. First, we require A-conjugacy among the directions p's. We want to avoid the bad behaviour of gradient descent when the condition of A is poor by imposing diversity. Another kind of conflicting goal is to achieve sufficient alignment between $p^{(k)}$ and $r^{(k-1)}$. Note that $r^{(k-1)}$ is exactly the direction in gradient descent. If they are not sufficently aligned, their dot product will be small, and the objective function will be decreased only very little.

Turns out these two considerations are simultaneously met with the following iteration rule, which turns out to be very simple

$$p^{(k)} = r^{(k-1)} + \beta_k p^{(k-1)}, \text{ where } \beta_k = -\frac{(p^{(k-1)})^T r^{(k-1)}}{(p^{(k-1)})^T Ap^{(k-1)}} \tag{19.10}$$

where $r^{(k-1)}$ is the previous residual and $p^{(k-1)}$ is the previous direction.

Convergence analysis. So what is the convergence analysis for conjugate gradient?

Theorem 19.2 *Conjugate gradient method satisfies*

$$\|x^{(k)} - x\|_A \le 2 \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^k \|x\|_A \tag{19.11}$$

where as before$\|x\|_A^2 = x^T Ax$ *and* $\kappa(A) = \lambda_1(A)/\lambda_n(A)$ *is the condition number of* A*. Further, it finds the exact solution* x *in at most* n *iterations.*

Proof: It is much much harder than gradient descent. Actually it is a classic topic. Just in recent years people have found different and new proofs which invokes Chebyshev polynomials and leads to other interesting algorithms. ■

We see that conjugate gradient too enjoys linear convergence but with a contraction factor that has a better dependence on $\kappa(A)$. To get $\|x^{(k)} - x\|_A \le \epsilon \|x^{(0))} - x\|_A$, we require $O(\sqrt{\kappa(A)} \log(1/\epsilon))$ iterations. For poorly conditioned A, the difference between $O(\kappa(A))$ and $O(\sqrt{\kappa(A)})$ can be a big deal.

19.5.4 Example

Here we conduct comparison of iterative methods for least squares problems: 100 i.i.d. standard Gaussian random instances with $n = 100, p = 20$. It is not a poorly conditioned system by the way.

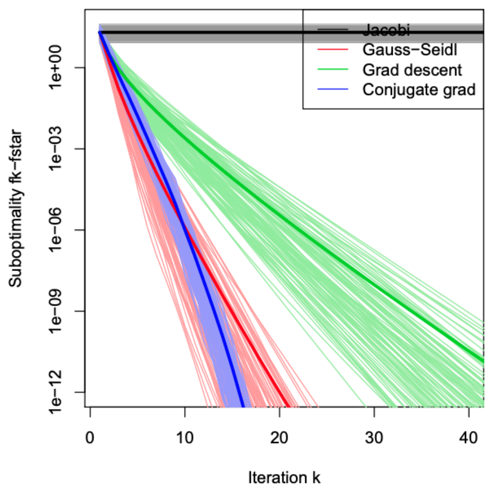


Figure 19.3: comparison of iterative methods for least squares problems: 100 i.i.d. standard Gaussian random instances with $n = 100, p = 20$.

Here it is fair to compare the methods w.r.t. their number of iterations because for arbitray one among them, each iteration costs $O(np)$. For Jacobi or Gauss-Seidl, we mean each full pass of the dimensions is an iteration.

As is shown in the Fig.19.3, Jacabi (black) is not converging. Gradient descent (green) can converge. Conjugate gradient (purple) and Gauss-Seidl share similar behavior. They both converge much faster than gradient descent even though it is not a poorly conditioned case. It looks that conjugate gradient does a little bit better job. However, we should note that, as a kind of coordinate descent, Gauss-Seidl can actually be applied to a wide range of problems, while conjugate gradient is limited to solving linear system.

19.6 Some advanced topics

There are many more interesting things to learn:

- Updating/downdating matrix factorizations
- Sparse matrix factorizations (SuiteSparse)
- Successive over-relaxation and acceleration
- Preconditioned conjugate gradient
- Laplacian (SDD) linear systems
- $\cdots$