CME 323: Distributed Algorithms and Optimization, Spring 2020
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matroid and Stanford.

Lecture 7, 4/21/2020. Scribed by Robin Brown.

7 Iterative Solutions for Solving Systems of Linear Equations

First we will introduce a number of methods for solving linear equations. These methods are extremely popular, especially when the problem is large such as those that arise from determining numerical solutions to linear partial differential equations.

The objective for solving a system of linear of equations is as follows. Let A be a full-rank matrix in $\mathbb{R}^{n \times n}$, and b be a vector in $\mathbb{R}^n$. The objective is to find x that satisfies

$$Ax = b.$$

We are guaranteed that x is unique because we assumed A to be invertible. The key observation is that the components of x can be solved for in terms of each other. Specifically, if $a_{ii} \neq 0$ and all of x_j, $j \neq i$ are known then we can solve for x_i as

$$x_i = -\frac{1}{a_{ii}} \left(\sum_{j \neq i} a_{ij} x_j - b_i \right).$$

If this is true for all i then we can solve for each x_i in parallel. Clearly, this cannot be the case (or we would have the solution!), however if have *estimates* for each component, we can solve for new estimates of the components in simultaneously. The algorithms introduced in this section work by iteratively computing estimates of the solution. If each estimate is closer to the true solution than the previous one, then we will converge to the true solution. Each algorithm is initialized with any $x(0) \in \mathbb{R}^n$, and the next iterates are computed as follows.

The Jacobi Algorithm applies this idea directly, by simply using the estimate of x at iteration t, x_t, to compute the estimate of x at time $t+1$, x_{t+1}. The iterates in the Jacobi algorithm are given as follows.

Jacobi Algorithm

$$x_i(t+1) = -\frac{1}{a_{ii}} \left(\sum_{j \neq i} a_{ij} x_j(t) - b_i \right).$$

The Gauss-Seidel algorithm boosts convergence by using information as soon as it is computed. Specifically, if $x_i(t+1)$ is computed before a processor begins computing $x_j(t+1)$, then the Gauss-Seidel algorithm uses $x_i(t+1)$ in place of $x_i(t)$. The iterates in the Gauss-Seidel algorithm are given as follows.

Gauss-Seidel Algorithm

$$x_i(t+1) = -\frac{1}{a_{ii}}\left(\sum_{j<i} a_{ij}x_j(t+1) + \sum_{j>i} a_{ij}x_j(t) - b_i\right).$$

The choice to update the coordinates in order is arbitrary, and in general, different update orders may produce substantially different results. We will see later how the order of updates can significantly impact how parallelizable the Gauss-Seidel algorithm is.

There are also relaxed versions of these algorithms that weight the previous iterate and the Jacobi and Gauss-Seidel updates respectively.

Jacobi Overrelaxation

$$x_i(t+1) = (1-\gamma)x_i(t) - \frac{\gamma}{a_{ii}}\left(\sum_{j\neq i} a_{ij}x_j(t) - b_i\right).$$

Successive Overrelaxation

$$x_i(t+1) = (1-\gamma)x_i(t) - \frac{\gamma}{a_{ii}}\left(\sum_{j<i} a_{ij}x_j(t+1) + \sum_{j>i} a_{ij}x_j(t) - b_i\right).$$

Richardson's Method Richardson's method is obtained by rewriting $Ax = b$ as $x = x - \gamma[Ax - b]$. The updates are then given by

$$x(t+1) = [I - \gamma A]x(t) + \gamma b$$

where γ is a scalar relaxation parameter. Richardson's method can also be executed in a Gauss-Seidel fashion (called RGS). Explicitly, the updates for the RGS algorithm are given by

$$x_i(t+1) = x_i(t) - \gamma\left[\sum_{i<j} a_{ij}x_j(t+1) + \sum_{j\geq i} a_{ij}x_j(t) - b_i\right]$$

Here we emphasize that the update order for the coordinates is variable. Another variant of the Richardson algorithm is obtained by rewriting $Ax = b$ as $x = x - B(Ax - b)$ where B is any invertible matrix. Then the iterates are given by

$$x(t+1) = x(t) - b\left[Ax(t) - b\right].$$

The Gauss-Seidel variant of this update is also possible.

7.1 Convergence of the Classical Iterative methods

We will now prove a general theorem that encompasses the convergence of the classical iterative methods.

Theorem 7.1 *Suppose $b \in \mathbb{R}^n$ and $A = M - N \in \mathbb{R}^{n \times n}$ is nonsingular. If M is nonsingular and the spectral radius of M satisfies $\rho(M^{-1}N) < 1$, then the iterates $x^{(k)}$ defines by $x^{(k+1)} = M^{-1}(Nx^{(k)} + b)$ converge to $x = A^{-1}b$ for any starting vector $x^{(0)}$. Additionally, $\left\|e^{(k)}\right\| \le \rho(M^{-1}N)^k \left\|e^{(0)}\right\|$. Consequently, we need $\log\left(\frac{\rho(M^{-1}N)}{e^{(0)}}\right)$ iterations to get an ϵ-approximate solution.*

Proof: *Let $e^{(k)} = x^{(k)} - x$ denotes the error in the kth iterate. Since $Mx = Nx + b$, it follows that $M(x^{(k+1)} - x) = N(x^{(k)} - x)$, and thus, the error in $x(k+1)$ is given by*

$$e^{(k+1)} = M^{-1}Ne^{(k)} = \left(M^{-1}N\right)^{k+1} e^{(0)}.$$

So, $\left\|e^{(k)}\right\| \le \rho(M^{-1}N)^k \left\|e^{(0)}\right\|$. Taking the log of both sides are rearranging gives the result. ■

For the Jacobi algorithm, M is a diagonal matrix with the same diagonal entries as A, and N is the negative of the off-diagonal entries of A. For the Gauss-Seidel algorithm, M is the diagonal, and subdiagonal entries of A, while N is the negative of all of the entries above the diagonal.

8 Parallel Implementation of Classical Iterative Methods

We now discuss how to parallelize the previously introduced iterative methods. The Jacobi and Jacobi overrelaxation algorithms are easily parallelized. The update for each component can be computed completely independently of each other. Suppose that the ith processor has access the ith row of A. Then the update can be calculated via the inner product between two vectors, and after each component is updated, processor i passes $x_i(t)$ to all of the other processors. Another implementation is one in which the ith processor has access to the ith column of A. After each update, processor i passes $a_{ji}x_i(t)$ to processor j. Each component update is computed by summing up all of the received values.

In contrast, the Gauss-Seidel and successive overrelaxation algorithms are not well-suited to parallel implementation. Unfortunately, the modifications made to boost convergence introduced an inherently sequential component to their implementation. As written, the compute $x_i(t+1)$, processor i needs to know the value of $x_j(t+1)$ for all $j < i$. Fortunately, when A is sparse (as is often the case when A arises from the discretization of a partial differential equation), we can use a coloring scheme to parallelize the updates.

8.1 Coloring

In the Gauss-Seidel algorithm, one might hope that if each coordinate update does not directly affect all of the other coordinates then they can be implemented in parallel. This is exactly the case. Precisely, notice that if $a_{ik} = 0$ for some $k < i$ then the updates

$$x_i(t+1) = -\frac{1}{a_{ii}}\left(\sum_{j<i} a_{ij}x_j(t+1) + \sum_{j>i} a_{ij}x_j(t) - b_i\right)$$

$$x_i(t+1) = -\frac{1}{a_{ii}}\left(\sum_{j<i, j\neq k} a_{ij}x_j(t+1) + \sum_{j>i} a_{ij}x_j(t) + a_{ik}x_k(t) - b_i\right)$$

are equivalent. We make this intuition precise via the following graph-theoretic notion of which Gauss-Seidel updates can be computed in parallel.

Given a dependency graph, $G = (V, E)$ representing the Gauss-Seidel iteration, a K-coloring is a map $C : V \to [K]$ that assigns a color $C(i)$ to each node in the graph.

Theorem 8.1 *There exists an ordering of the variables such that the Gauss-Seidel algorithm can be performed in K parallel steps if and only if there exists a K-coloring of the dependency graph where no positive cycle is entirely the same color.*

Proof: ($\Rightarrow$) *Consider an ordering of the variables with which the Gauss-Seidel iteration takes K parallel steps. We define our coloring map by $C(i) = k$ if node i is updated at the kth step. Then given any positive cycle $i_1, \ldots, i_m$, let i_l be the node that is updated first out of all $i_1, \ldots, i_m$. Since $(i_l, i_{l+1}) \in E$ this means $x_{i_{l+1}}$ depends on x_{i_l}, and the two variables cannot be updated in the same step. Consequently, they cannot be assigned the same color, and the cycle $i_1, \ldots, i_m$ has more than one color.*

($\Leftarrow$) *We will use the result that if a graph is a DAG, there there exists an ordering on the nodes such that if (i, j) is an edge, then node j comes before node i in the ordering.*

Now assume that there exists a K-coloring of $G = (V, E)$ such that no positive cycle is entirely the same color. We define the subgraphs G_k of G by only keeping nodes of the color k and the edges joining them. Then, by assumption, each of the subgraphs G_k is a DAG, so by the previous result, there is a topological ordering of each of these subgraphs. Then we order the nodes of G in increasing color order, where the ordering of nodes of the same color depends on the topological ordering of their associated subgraph. Consider the Gauss-Seidel update according to this ordering. Let nodes i and j have the same color k. Then if $(i, j) \in E$ and $(j, i) \in E$, then clearly x_i and x_j cannot be updated in parallel. It is not possible for $(i, j) \in E$ and $(j, i) \in E$ if nodes i and j have the same color k since each G_k is acyclic. If $(i, j) \in E$ and $(j, i) \notin E$, then node j is updated before node i, and so the computation of $x_j(t + 1)$ only requires the value of $x_i(t)$ and not $x_i(t + 1)$. The case where $(i, j) \notin E$ and $(j, i) \in E$ follows similarly. Therefore, every node of the same color can be updated in parallel, proving the result. ■

It's important to note that the dependency graph for the Gauss-Seidel updates is not necessarily a

DAG. To illustrate this, we consider an example where

$$\begin{aligned}
x_1(t+1) &= f_1(x_1(t), x_3(t)) \\
x_2(t+1) &= f_2(x_1(t), x_2(t)) \\
x_3(t+1) &= f_3(x_2(t), x_3(t), x_4(t)) \\
x_4(t+1) &= f_4(x_2(t), x_4(t))
\end{aligned}$$

Figure 8.1 depicts the dependency graph for this problem.

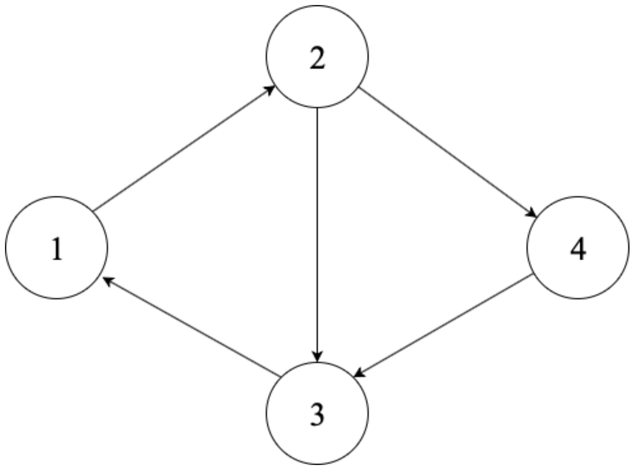
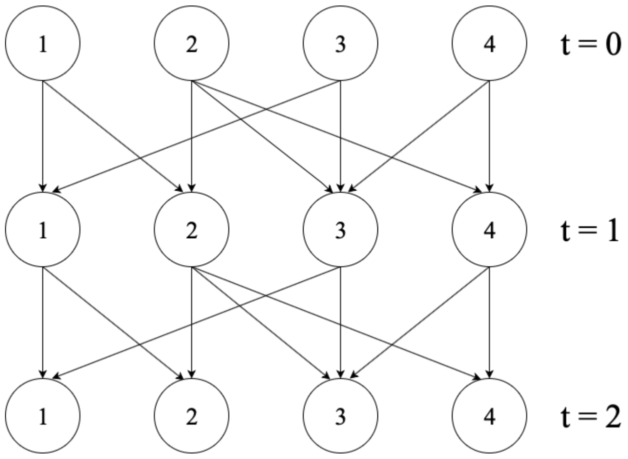


Figure 8.1 shows the DAG representing the Jacobi updates for the first two times steps.



Figures 8.1 and 8.1 show the DAGs representing the Gauss-Seidel updates for two different update orders. In figure 8.1, the variables are updated in order $x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_4$, whereas in figure 8.1 the variables are updated in order $x_1 \rightarrow x_3 \rightarrow x_4 \rightarrow x_2$. The first ordering requires 3 parallel steps while the second only requires 2.

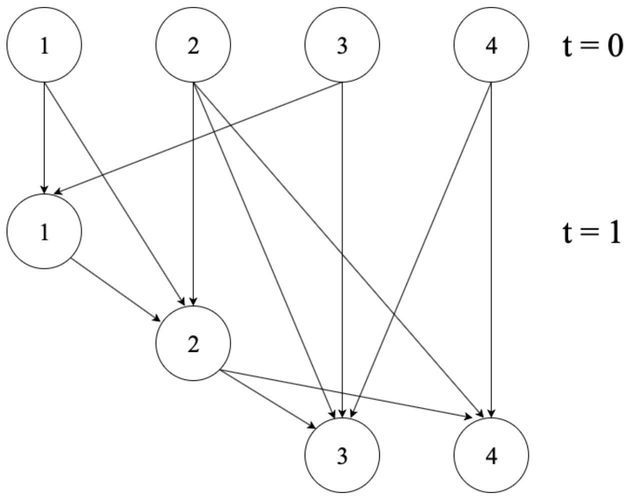


Figure 1: Gauss-Seidel updates with ordering $x_1 \to x_2 \to x_3 \to x_4$

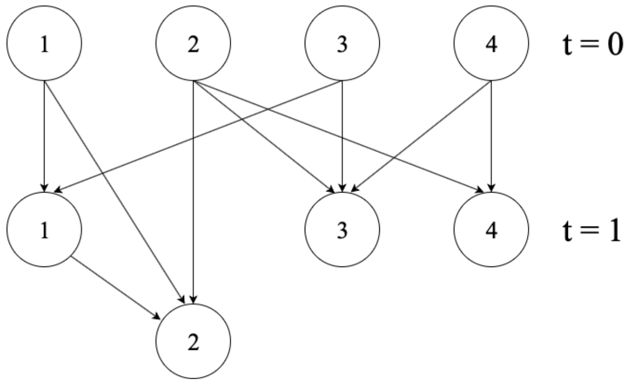


Figure 2: Gauss-Seidel updates with ordering $x_1 \to x_3 \to x_4 \to x_2$

9 Unconstrained Optimization

We will now show that the iterative algorithms for solving linear systems can be generalized to develop optimization algorithms.

In unconstrained optimization, the objective is to minimize some function $F : \mathbb{R}^n \to \mathbb{R}$. If F is continuously differentiable, and x^* is a minimizer of F, then $\nabla F(x^*) = 0$. Consequently, minimizing F is closely related to solving the nonlinear system of equations $\nabla F(x^*) = 0$. In general, without the appropriate guarantees of convexity, $\nabla F(x) = 0$ does not guarantee that x is a global minimizer of F. However, assuring that a critical point is a global minimizer is often intractable so many algorithms will settle for a critical point instead.

We will motivate the derivation of these algorithms by noticing that solving $Ax = b$ is equivalent to minimizing $F(x) = \frac{1}{2}x^T Ax - x^T b$. Then the gradient and Hessian of F are given by $\nabla F(x) = Ax - b$ and $\nabla^2 F(x) = A$. Then, the following algorithms can be interpeted as generalizations of the Jacobi overrelaxation, successive overrelaxation, and Richardson's algorithm respectively.

Jacobi Algorithm

$$x(t+1) = x(t) - \gamma[diag(\nabla^2 F(x(t)))]^{-1}\nabla F(x(t))$$

where $diag(\nabla^2 F(x(t)))$ is the diagonal matrix with the same diagonal elements as $\nabla^2 F(x(t))$

Gauss Seidel Algorithm

$$x_i(t+1) = x_(t) - \gamma \frac{\nabla_i F(\hat{x}(i,t)}{\nabla^2_{ii} F(\hat{x}(i,t)}$$

where $\hat{x}(i,t) = (x_1(t+1), \ldots, x_{i-1}(t+1), x_i(t), \ldots, x_n(t))$ is the most recent update of x

Gradient Algorithm Richardson's method can similarly be generalized as

$$x(t+1) = x(t) - \gamma F(x(t)).$$

And the Gauss-Seidel variant of the gradient algorithm is given by

$$x_i(t+1) = x_i(t) - \gamma \nabla_i F(\hat{x}(i,t)), \quad i = 1, \ldots, n$$

For a fixed $x \in \mathbb{R}^n$ where $\nabla F(x) \neq 0$, if $v \in \mathbb{R}^n$ is a vector such that $v^T \nabla F(x) < 0$ then v is said to be a descent direction. This is because since F is continuously differentiable, there exists a positive γ small enough such that $F(x + \gamma v) < F(x)$. Any algorithm that computes the next iterate by updating $x(t)$ along a descent direction is said to be a *descent algorithm.* The Jacobi, Gauss-Seidel, and gradient algorithms are all descent algorithms. They are generalized by the following scaled gradient algorithm

Scaled Gradient Algorithms

$$x(t+1) = x(t) - \gamma(D(t))^{-1}\nabla F(x(t))$$

where $D(t)$ is a scaling matrix. In practice D is often chosen to be diagonal so its inverse is just the diagonal matrix with entries $\frac{1}{D_{ii}}$. Not only is its inverse easy to calculate, but it is also more straightforward to implement in parallel.

9.1 Nonlinear Algorithms

The algorithms presented up until this point are called *linear algorithms* because each of the updates is a linear function of $\nabla F(x)$. *Nonlinear* or *coordinate descent* algorithms work by fixing all components of x but x_i and minimizing $F(x)$ with respect to only x_i.

In the nonlinear Jacobi algorithm, the minimizations with respect to each of the x_i are carried out simultaneously. Explicitly, the updates are given by

$$x_i(t+1) = \arg\min_{x_i} F(x_1(t), \ldots, x_{i-1}(t), x_i, x_{i+1}(t), \ldots x_n(t)).$$

Similarly, the Gauss-Seidel updates are carried out by using the most recent information as it is computed. Explicitly, the Gauss-Seidel updates are given by

$$x_i(t+1) = \min_{x_i} F(x_1(t+1), \ldots, x_{i-1}(t+1), x_i, x_{i+1}(t), \ldots x_n(t)).$$

9.2 Parallel Implementation

Just as before, the parallel implementation of the Jacobi, and gradient algorithms are straightforward. We can assign the computation of $x_i(t)$ to the ith processor. After each $x_i(t)$ is computed, its value is communicated only to the processors which require it. In particular, the ith processor needs to know the current value of $x_j(t)$ if $\nabla_i F$ or $\nabla^2_{ii} F$ depends on x_j. For many problems arising in practice, both $\nabla_i F$ and $\nabla^2_{ii} F$ are sparse; this can be leveraged to drastically reduce the communication requirement of the algorithms.

In general, the Gauss-Seidel algorithms are unsuitable for parallel implementation except when the dependence graph is sparse. Then, the same coloring scheme technique can be applied to parallelize the computation.

One might recall that a common technique to boost convergence of optimization algorithms is to vary the step-size $\gamma(t)$ with each iteration such that $F(x(t) - \gamma \nabla F(x(t)))$ is minimized. Unfortunately, this minimization step is not amenable to parallelization.

10 Constrained Optimization

We will now consider the problem of constrained optimization. Here, our objective is to

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & F(x) \\ \text{subject to} \quad & x \in \chi \end{aligned} \tag{1}$$

We will assume $F : \mathbb{R}^n \to \mathbb{R}$ to be continuously differentiable, and χ to be nonempty, closed, and convex. The necessary condition for $x \in \chi$ to be optimal is that $(y - x)^T \nabla F(x) \geq 0$ for all $y \in \chi$. If F is convex over χ then this condition is also sufficient.

10.1 Projected Gradient Descent

We will want to apply the descent algorithms of section 9 for constrained optimization. However, we will not be able to apply them directly. This is because even if an iterate is feasible i.e., $x(t) \in \chi$, it is not guaranteed that $x(t+1)$ is feasible. We will remedy this by simply projecting $x(t+1)$ back into the feasible set χ. In particular, we will define the projection operation as

$$\Pi_\chi(x) = \arg\min_{z \in \chi} \|x - z\|$$

and the iterates in the projected gradient algorithm are given by

$$x(t+1) = \Pi_\chi\left(x(t) - \gamma \nabla F(x(t))\right)$$

Now we just need to show that our projection step is well-defined and the progress we have made in our descent step is not completely undone by the projection step. This is will follow from the projection theorem:

Theorem 10.1 (Projection Theorem) *1. For every $x \in \mathbb{R}^n$ there exists a unique $z \in \chi$ that minimizes $\|x - z\|$ over all $z \in \chi$ (denoted as $\Pi_\chi(x)$).*

2. *For some $x \in \mathbb{R}^n$, $Z \in \chi$ is equal to $\Pi_\chi(x)$ is and only if $(y-z)^T(x-z) \leq 0$ for all $y \in \chi$.*

3. *The mapping $\Pi_\chi : \mathbb{R}^n \to \chi$ is continuous and nonexpansive. In other words,*

$$\|\Pi_\chi(x) - \Pi_\chi(y)\|_2 \leq \|x-y\|_2$$

for all $x, y \in \mathbb{R}^n$.

10.2 Parallel Implementation

The gradient projection algorithm is generally not well-suite to parallelization due to the projection step. One case where it is, though, is when the feasible set χ can be represented as the Cartesian product of constraint sets for the individual components of x i.e., $\chi = \prod_{i=1}^{n}[l_i, u_i]$. In this case, the projection onto χ is straightforward—if processor i is responsible for computing x_i, then it can simply project x_i onto $[l_i, u_i]$. This approach can also be generalized when the constraint set is a Cartesian product of sets i.e., $\chi = \prod_{i=1}^{m}\chi_i$. By viewing $\mathbb{R}^n$ as the Cartesian product of spaces $\mathbb{R}^{n_i}$, where $n_1 + \ldots + n_m = n$ and each χ_i is a closed convex subset of $\mathbb{R}^{n_i}$, it is apparent that projecting x onto χ is equivalent to projecting the appropriate components of x onto χ_i individually. In other words $\Pi_\chi(x) = \prod_{i=1}^{m}\Pi_{chi_i}(x^{(i)})$, where here we let $x^{(i)}$ denote the appropriate components of x corresponding to χ_i. For example, if $\chi = \{x \in \mathbb{R}^4 | x_1 + x_2 = 0,\ x_3 + x_4 = 0\}$, then we can express $\chi = \{x \in \mathbb{R}^2 | x_1 + x_2 = 0\} \otimes \{x \in \mathbb{R}^2 | x_3 + x_4 = 0\}$, and $x^{(1)} = (x_1, x_2)$, $x^{(2)} = (x_3, x_4)$.

A similar discussion also applies to the parallel implementations of the scaled gradient algorithm. In general, we cannot expect to be able to compute $x(t+1) = x(t) - \gamma(D(t))^{-1}\nabla F(x(t))$ in a distributed manner. However, if $(D(t))^{-1}$ has a "nice" structure, it is possible. We briefly discussed the case where $D(t)$ is diagonal, and thus $(D(t))^{-1}$ is diagonal and each components updates can be computed in parallel. In general, if there is a permutation, represented by matrix P, where $P(D(t))^{-1}$ is block diagonal, then the components in the same block (after permutation) need to be updated together, but those that are not in the same block can be updated independently of each other.

10.3 Distributed Nonlinear Algorithms

If we assume χ to be a Cartesian product, then it makes sense for us to consider the projected nonlinear Jacobi and Gauss-Seidel algorithms. They operate simply by restricting the the minimization to the feasible sets for each component. The updates are given as follows:

Jacobi

$$x_i(t+1) = \min_{x_i \in [l_i, u_i]} F(x_1(t), \ldots, x_{i-1}(t), x_i, x_{i+1}(t), \ldots x_n(t)).$$

Gauss-Seidel

$$x_i(t+1) = \min_{x_i \in [l_i, u_i]} F(x_1(t+1), \ldots, x_{i-1}(t+1), x_i, x_{i+1}(t), \ldots x_n(t)).$$

10.4 Parallelization by Decomposition

Previously we showed that a number of descent algorithms can be parallelized. However, these methods are not always applicable, especially for constrained optimization (which require the constraint set to be the Cartesian product of constraints on individual components). We will now explore how specific problem structure can be exploited to derive a parallel solution algorithm by find a suitable transformation of the problem. Oftentimes the dual optimization problem is much more amenable to parallel implementation than the original problem. We will illustrate this idea with several examples.

10.5 Quadratic Programming

The objective of a quadratic programming problem is to solve a problem of the form

$$\begin{array}{cl} \underset{x \in \mathbb{R}^n}{\text{minimize}} & \frac{1}{2}x^TQx - b^Tx \\ \text{subject to} & Ax \leq c \end{array} \tag{2}$$

where $Q \in \mathbb{R}^{n\times n}$ is positive definite, $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^n$, and $c \in \mathbb{R}^m$ are all known. The dual of this problem is given by

$$\begin{array}{cl} \underset{u \in \mathbb{R}^m}{\text{minimize}} & \frac{1}{2}u^T(AQ^{-1}A^T)u + (c - AQ^{-1}b)^Tu \\ \text{subject to} & u \geq 0 \end{array} \tag{3}$$

If u^* is the optimal solution to problem (3), then the optimal solution x^* can be recovered by the relation $x^* = Q^{-1}(b - A^Tu^*)$. Notice that while the constraint set defined by $\{x|Ax \leq c\}$ cannot generally be expressed as the Cartesian product of simpler sets, $u \geq 0$ certainly can, making the methods we have previously discussed suitable for solving the dual problem. In particular, we will consider the non-linear Jacobi algorithm.

Let $D(u) = \frac{1}{2}u^TPu+r^Tu$, with $P = AQ^{-1}A^T$,and $r = c-AQ^{-1}b$ be the dual objective function. Note that $\nabla_j D(u) = r_j + \sum_{k=1}^m p_{jk}u_k$. Because the dual objective is convex, its minimizer can be found by finding u such that $\nabla D(u) = 0$. Using the expression we derived for $\nabla_j D(u)$, we see that

$$\arg\min_{u_j} = \tilde{u}_j = -\frac{\gamma}{p_{jj}}\left(r_j + \sum_{k\neq j} p_{jk}u_k\right)$$

Then, taking into account the nonnegativity constraint, $u_j = \max\{0, \tilde{u}_j\}$. Our iterates are then given by

$$u_j(t+1) = \max\left\{0, u_j(t) - \frac{\gamma}{p_{jj}}\left(r_j + \sum_{k=1}^m p_{jk}u_{k(t)}\right)\right\}.$$

Notice that each of components can be updated in parallel.

10.6 Separable Strictly Convex Programming

Suppose that the space $\mathbb{R}^n$ is represented as the Cartesian product of spaces $\mathbb{R}^{n_i}$, $i = 1, \ldots, m$ where $n_1 + \ldots + n_m = n$, and consider the problem

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & \sum_{i=1}^{m} F_i(x^{(i)}) \\ \text{subject to} \quad & e_j^T x = s_j, \quad j = 1, \ldots, r, \\ & x^{(i)} \in P_i, \quad i = 1, \ldots, m \end{aligned} \tag{4}$$

Where $F_i : \mathbb{R}^{n_i} \to \mathbb{R}$ are strictly convex, $x^{(i)}$ are the appropriate components of x, e_j are given vectors in $\mathbb{R}^n$, s_j ar es alres, and P_i are bounded polyhedral subsets of $\mathbb{R}^{n_i}$. Then, the dual problem is given by

$$\underset{p \in \mathbb{R}^r}{\text{maximize}} \quad q(p) \tag{5}$$

where,

$$q(p) = \min_{x^{(i)} \in \mathbb{R}^{n_i}} \left(\sum_{i=1}^{m} F_i(x^{(i)}) + \sum_{j=1}^{n} p_j(e_j^T x - s_j) \right) \tag{6}$$

This is now separable

$$q_i(p) = \min_{x_i \in P_i} \left(F_i(x^{(i)) + \sum_{j=1}^{r} p_j \hat{e}_{ji}^T x_i} \right) \tag{7}$$

where $\hat{e}_{ji}$ denotes the appropriate components of e_j corresponding to $x^{(i)}$. The evaluation of the dual function is amenable to parallelization with each separate processor computing a component $q_i(p)$ of $q(p)$.