16 Interior Point 1: Linear Programs

Lab Objective: *For decades after its invention, the Simplex algorithm was the only competitive method for linear programming. The past 30 years, however, have seen the discovery and widespread adoption of a new family of algorithms that rival–and in some cases outperform–the Simplex algorithm, collectively called* Interior Point methods*. One of the major shortcomings of the Simplex algorithm is that the number of steps required to solve the problem can grow exponentially with the size of the linear system. Thus, for certain large linear programs, the Simplex algorithm is simply not viable. Interior Point methods offer an alternative approach and enjoy much better theoretical convergence properties. In this lab we implement an Interior Point method for linear programs, and in the next lab we will turn to the problem of solving quadratic programs.*

Introduction

Recall that a linear program is a constrained optimization problem with a linear objective function and linear constraints. The linear constraints define a set of allowable points called the *feasible region*, the boundary of which forms a geometric object known as a *polytope*. The theory of convex optimization ensures that the optimal point for the objective function can be found among the vertices of the feasible polytope. The Simplex Method tests a sequence of such vertices until it finds the optimal point. Provided the linear program is neither unbounded nor infeasible, the algorithm is certain to produce the correct answer after a finite number of steps, but it does not guarantee an efficient path along the polytope toward the minimizer. Interior point methods do away with the feasible polytope and instead generate a sequence of points that cut through the interior (or exterior) of the feasible region and converge iteratively to the optimal point. Although it is computationally more expensive to compute such interior points, each step results in significant progress toward the minimizer. See Figure 16.1 for an example of a path using an Interior Point algorithm. In general, the Simplex Method requires many more iterations (though each iteration is less expensive computationally).

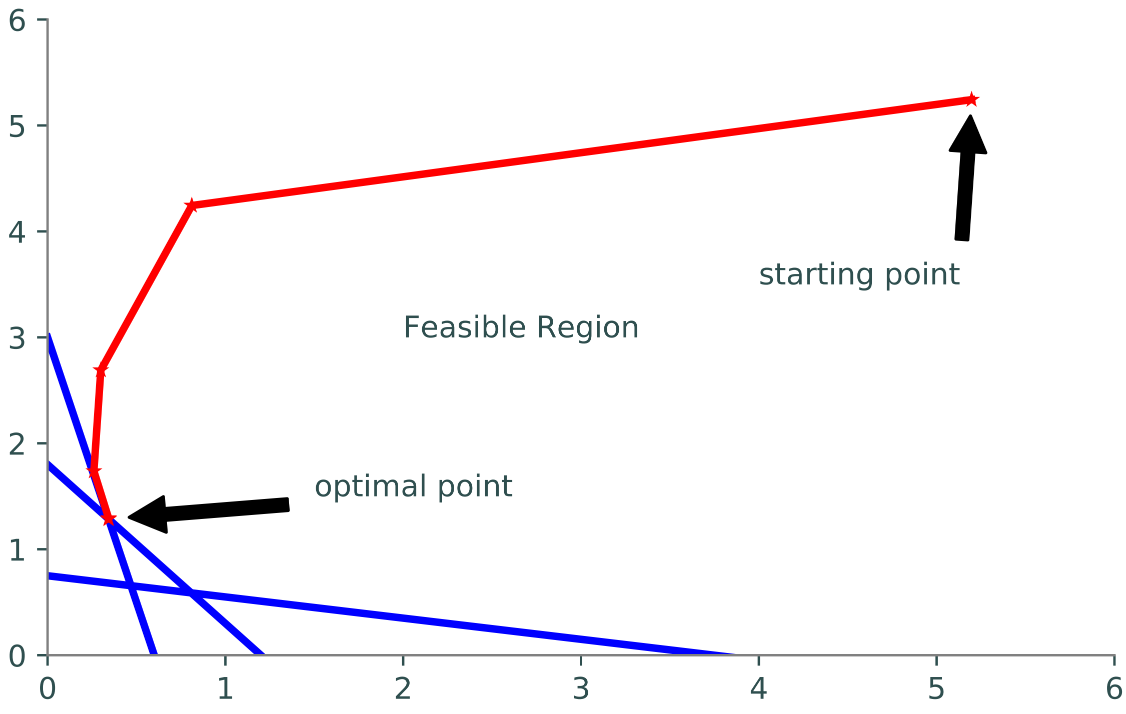

Figure 16.1: A path traced by an Interior Point algorithm.

Primal-Dual Interior Point Methods

Some of the most popular and successful types of Interior Point methods are known as Primal-Dual Interior Point methods. Consider the following linear program:

$$\begin{aligned} \text{minimize} \quad & \mathbf{c}^\mathsf{T}\mathbf{x} \\ \text{subject to} \quad & A\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \succeq \mathbf{0}. \end{aligned}$$

Here, $\mathbf{x}, \mathbf{c} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^m$, and $A \in \mathbb{R}^{m \times n}$ with full row rank. This is the *primal* problem, and its *dual* takes the form:

$$\begin{aligned} \text{maximize} \quad & \mathbf{b}^\mathsf{T}\boldsymbol{\lambda} \\ \text{subject to} \quad & A^\mathsf{T}\boldsymbol{\lambda} + \boldsymbol{\mu} = \mathbf{c} \\ & \boldsymbol{\mu}, \boldsymbol{\lambda} \succeq \mathbf{0}, \end{aligned}$$

where $\boldsymbol{\lambda} \in \mathbb{R}^m$ and $\boldsymbol{\mu} \in \mathbb{R}^n$.

KKT Conditions

The theory of convex optimization gives us necessary and sufficient conditions for the solutions to the primal and dual problems via the Karush-Kuhn-Tucker (KKT) conditions. The Lagrangian for the primal problem is as follows:

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \mathbf{c}^\mathsf{T}\mathbf{x} + \boldsymbol{\lambda}^\mathsf{T}(\mathbf{b} - A\mathbf{x}) - \boldsymbol{\mu}^\mathsf{T}\mathbf{x}$$

The KKT conditions are

$$\begin{aligned} A^\mathsf{T}\boldsymbol{\lambda} + \boldsymbol{\mu} &= \mathbf{c} \\ A\mathbf{x} &= \mathbf{b} \\ x_i\mu_i &= 0, \quad i = 1, 2, \ldots, n, \\ \mathbf{x}, \boldsymbol{\mu} &\succeq 0. \end{aligned}$$

It is convenient to write these conditions in a more compact manner, by defining an almost-linear function F and setting it equal to zero:

$$F(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) := \begin{bmatrix} A^\mathsf{T}\boldsymbol{\lambda} + \boldsymbol{\mu} - \mathbf{c} \\ A\mathbf{x} - \mathbf{b} \\ M\mathbf{x} \end{bmatrix} = \mathbf{0},$$
$$(\mathbf{x}, \boldsymbol{\mu} \succeq \mathbf{0}),$$

where $M = \operatorname{diag}(\mu_1, \mu_2, \ldots, \mu_n)$. Note that the first row of F is the KKT condition for dual feasibility, the second row of F is the KKT condition for the primal problem, and the last row of F accounts for complementary slackness.

Problem 1. Define a function `interiorPoint()` that will be used to solve the complete interior point problem. This function should accept A, $\mathbf{b}$, and $\mathbf{c}$ as parameters, along with the keyword arguments `niter=20` and `tol=1e-16`. The keyword arguments will be used in a later problem.

In the next few problems, you will be writing functions within this function to solve the interior point problem one step at a time.

For this problem, within the `interiorPoint()` function, write a function for the vector-valued function F described above. This function should accept $\mathbf{x}$, $\boldsymbol{\lambda}$, and $\boldsymbol{\mu}$ as parameters and return a 1-dimensional NumPy array with $2n + m$ entries.

Search Direction

A Primal-Dual Interior Point method is a line search method that starts with an initial guess $(\mathbf{x}_0^\mathsf{T}, \boldsymbol{\lambda}_0^\mathsf{T}, \boldsymbol{\mu}_0^\mathsf{T})$ and produces a sequence of points that converge to $(\mathbf{x}^{*\mathsf{T}}, \boldsymbol{\lambda}^{*\mathsf{T}}, \boldsymbol{\mu}^{*\mathsf{T}})$, the solution to the KKT equations and hence the solution to the original linear program. The constraints on the problem make finding a search direction and step length a little more complicated than for the unconstrained line search we have studied previously.

In the spirit of Newton's Method, we can form a linear approximation of the system $F(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \mathbf{0}$ centered around our current point $(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu})$, and calculate the direction $(\triangle\mathbf{x}^\mathsf{T}, \triangle\boldsymbol{\lambda}^\mathsf{T}, \triangle\boldsymbol{\mu}^\mathsf{T})$ in which to step to set the linear approximation equal to $\mathbf{0}$. This equates to solving the linear system:

$$DF(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) \begin{bmatrix} \triangle\mathbf{x} \\ \triangle\boldsymbol{\lambda} \\ \triangle\boldsymbol{\mu} \end{bmatrix} = -F(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) \tag{16.1}$$

Here $DF(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu})$ denotes the total derivative matrix of F. We can calculate this matrix block-wise by obtaining the partial derivatives of each block entry of $F(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu})$ with respect to $\mathbf{x}$, $\boldsymbol{\lambda}$, and $\boldsymbol{\mu}$, respectively. We thus obtain:

$$DF(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \begin{bmatrix} 0 & A^\mathsf{T} & I \\ A & 0 & 0 \\ M & 0 & X \end{bmatrix}$$

where $X = \operatorname{diag}(x_1, x_2, \ldots, x_n)$.

Unfortunately, solving Equation 16.1 often leads to a search direction that is too greedy. Even small steps in this direction may lead the iteration out of the feasible region by violating one of the constraints. To remedy this, we define the *duality measure* ν[1] of the problem:

$$\nu = \frac{\mathbf{x}^\mathsf{T}\boldsymbol{\mu}}{n}$$

The idea is to use Newton's method to identify a direction that strictly decreases ν. Thus instead of solving Equation 16.1, we solve:

$$DF(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) \begin{bmatrix} \triangle\mathbf{x} \\ \triangle\boldsymbol{\lambda} \\ \triangle\boldsymbol{\mu} \end{bmatrix} = -F(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) + \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \sigma\nu\mathbf{e} \end{bmatrix} \tag{16.2}$$

where $\mathbf{e} = (1, 1, \ldots, 1)^\mathsf{T}$ and $\sigma \in [0, 1)$ is called the *centering parameter*. The closer σ is to 0, the more similar the resulting direction will be to the plain Newton direction. The closer σ is to 1, the more the direction points inward to the interior of the of the feasible region.

Problem 2. Within `interiorPoint()`, write a subroutine to compute the search direction $(\triangle\mathbf{x}^\mathsf{T}, \triangle\boldsymbol{\lambda}^\mathsf{T}, \triangle\boldsymbol{\mu}^\mathsf{T})$ by solving Equation 16.2. Use $\sigma = \frac{1}{10}$ for the centering parameter.

Note that only the last block row of DF will need to be changed at each iteration (since M and X depend on $\boldsymbol{\mu}$ and $\mathbf{x}$, respectively). Use the functions `lu_factor()` and `lu_solve()` from the `scipy.linalg` module to solving the system of equations efficiently.

Step Length

Now that we have our search direction, it remains to choose our step length. We wish to step nearly as far as possible without violating the problem's constraints, thus remaining in the interior of the feasible region. First, we calculate the maximum allowable step lengths for $\mathbf{x}$ and $\boldsymbol{\mu}$, respectively:

$$\alpha_{\max} = \min\{-\mu_i/\triangle\mu_i \mid \triangle\mu_i < 0\}$$
$$\delta_{\max} = \min\{-x_i/\triangle x_i \mid \triangle x_i < 0\}$$

If all values of $\triangle\mu$ are nonnegative, let $\alpha_{\max} = 1$. Likewise, if all values of $\triangle x$ are nonnegative, let $\delta_{\max} = 1$. Next, we back off from these maximum step lengths slightly:

$$\alpha = \min(1, 0.95\alpha_{\max})$$
$$\delta = \min(1, 0.95\delta_{\max}).$$

[1]ν is the Greek letter for n, pronounced "nu."

These are our final step lengths. Thus, the next point in the iteration is given by:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \delta \triangle \mathbf{x}_k$$
$$(\boldsymbol{\lambda}_{k+1}, \boldsymbol{\mu}_{k+1}) = (\boldsymbol{\lambda}_k, \boldsymbol{\mu}_k) + \alpha(\triangle \boldsymbol{\lambda}_k, \triangle \boldsymbol{\mu}_k).$$

Problem 3. Within `interiorPoint()`, write a subroutine to compute the step size after the search direction has been computed. Avoid using loops when computing $\alpha_{\max}$ and $\delta_{\max}$ (use masking and NumPy functions instead).

Initial Point

Finally, the choice of initial point $(\mathbf{x}_0, \boldsymbol{\lambda}_0, \boldsymbol{\mu}_0)$ is an important, nontrivial one. A naïvely or randomly chosen initial point may cause the algorithm to fail to converge. The following function will calculate an appropriate initial point.

```python
def starting_point(A, b, c):
    """Calculate an initial guess to the solution of the linear program
    min c\trp  x, Ax = b, x>=0.
    Reference: Nocedal and Wright, p. 410.
    """
    # Calculate x, lam, mu of minimal norm satisfying both
    # the primal and dual constraints.
    B = la.inv(A @ A.T))
    x = A.T @ B @ b
    lam = B @ A @ c
    mu = c - (A.T @ lam)

    # Perturb x and s so they are nonnegative.
    dx = max((-3./2)*x.min(), 0)
    dmu = max((-3./2)*mu.min(), 0)
    x += dx*np.ones_like(x)
    mu += dmu*np.ones_like(mu)

    # Perturb x and mu so they are not too small and not too dissimilar.
    dx = .5*(x*mu).sum()/mu.sum()
    dmu = .5*(x*mu).sum()/x.sum()
    x += dx*np.ones_like(x)
    mu += dmu*np.ones_like(mu)

    return x, lam, mu
```

Problem 4. Complete the implementation of `interiorPoint()`.

Use the function `starting_point()` provided above to select an initial point, then run the iteration `niter` times, or until the duality measure is less than `tol`. Return the optimal point $\mathbf{x}^*$ and the optimal value $\mathbf{c}^\mathsf{T}\mathbf{x}^*$.

The duality measure ν tells us in some sense how close our current point is to the minimizer. The closer ν is to 0, the closer we are to the optimal point. Thus, by printing the value of ν at each iteration, you can track how your algorithm is progressing and detect when you have converged.

To test your implementation, use the following code to generate a random linear program, along with the optimal solution.

```python
def randomLP(j, k):
    """Generate a linear program min c\trp  x s.t. Ax = b, x>=0.
    First generate m feasible constraints, then add
    slack variables to convert it into the above form.
    Inputs:
        j (int >= k): number of desired constraints.
        k (int): dimension of space in which to optimize.
    Outputs:
        A ((j, j+k) ndarray): Constraint matrix.
        b ((j,) ndarray): Constraint vector.
        c ((j+k,), ndarray): Objective function with j trailing 0s.
        x ((k,) ndarray): The first 'k' terms of the solution to the LP.
    """
    A = np.random.random((j,k))*20 - 10
    A[A[:,-1]<0] *= -1
    x = np.random.random(k)*10
    b = np.zeros(j)
    b[:k] = A[:k,:] @ x
    b[k:] = A[k:,:] @ x + np.random.random(j-k)*10
    c = np.zeros(j+k)
    c[:k] = A[:k,:].sum(axis=0)/k
    A = np.hstack((A, np.eye(j)))
    return A, b, -c, x
```

```python
>>> j, k = 7, 5
>>> A, b, c, x = randomLP(j, k)
>>> point, value = interiorPoint(A, b, c)
>>> np.allclose(x, point[:k])
True
```

Least Absolute Deviations (LAD)

We now return to the familiar problem of fitting a line (or hyperplane) to a set of data. We have previously approached this problem by minimizing the sum of the squares of the errors between the data points and the line, an approach known as *least squares*. The least squares solution can be obtained analytically when fitting a linear function, or through a number of optimization methods (such as Conjugate Gradient) when fitting a nonlinear function.

The method of *least absolute deviations* (LAD) also seeks to find a best fit line to a set of data, but the error between the data and the line is measured differently. In particular, suppose we have a set of data points $(y_1, \mathbf{x}_1), (y_2, \mathbf{x}_2), \ldots, (y_m, \mathbf{x}_m)$, where $y_i \in \mathbb{R}$, $\mathbf{x}_i \in \mathbb{R}^n$ for $i = 1, 2, \ldots, m$. Here, the $\mathbf{x}_i$ vectors are the *explanatory variables* and the y_i values are the *response variables*, and we assume the following linear model:

$$y_i = \boldsymbol{\beta}^\mathsf{T}\mathbf{x}_i + b, \qquad i = 1, 2, \ldots, m,$$

where $\boldsymbol{\beta} \in \mathbb{R}^n$ and $b \in \mathbb{R}$. The error between the data and the proposed linear model is given by

$$\sum_{i=1}^{n} |\boldsymbol{\beta}^\mathsf{T}\mathbf{x}_i + b - y_i|,$$

and we seek to choose the parameters $\boldsymbol{\beta}, b$ so as to minimize this error.

Advantages of LAD

The most prominent difference between this approach and least squares is how they respond to outliers in the data. Least absolute deviations is robust in the presence of outliers, meaning that one (or a few) errant data points won't severely affect the fitted line. Indeed, in most cases, the best fit line is guaranteed to pass through at least two of the data points. This is a desirable property when the outliers may be ignored (perhaps because they are due to measurement error or corrupted data). Least squares, on the other hand, is much more sensitive to outliers, and so is the better choice when outliers cannot be dismissed. See Figure 16.2.

While least absolute deviations is robust with respect to outliers, small horizontal perturbations of the data points can lead to very different fitted lines. Hence, the least absolute deviations solution is less stable than the least squares solution. In some cases there are even infinitely many lines that minimize the least absolute deviations error term. However, one can expect a unique solution in most cases.

The least absolute deviations solution arises naturally when we assume that the residual terms $\boldsymbol{\beta}^\mathsf{T}\mathbf{x}_i + b - y_i$ have a particular statistical distribution (the Laplace distribution). Ultimately, however, the choice between least absolute deviations and least squares depends on the nature of the data at hand, as well as your own good judgment.

LAD as a Linear Program

We can formulate the least absolute deviations problem as a linear program, and then solve it using our interior point method. For $i = 1, 2, \ldots, m$ we introduce the artificial variable u_i to take the place of the error term $|\boldsymbol{\beta}^\mathsf{T}\mathbf{x}_i + b - y_i|$, and we require this variable to satisfy $u_i \geq |\boldsymbol{\beta}^\mathsf{T}\mathbf{x}_i + b - y_i|$. This constraint is not yet linear, but we can split it into an equivalent set of two linear constraints:

$$\begin{aligned} u_i &\geq \boldsymbol{\beta}^\mathsf{T}\mathbf{x}_i + b - y_i, \\ u_i &\geq y_i - \boldsymbol{\beta}^\mathsf{T}\mathbf{x}_i - b. \end{aligned}$$

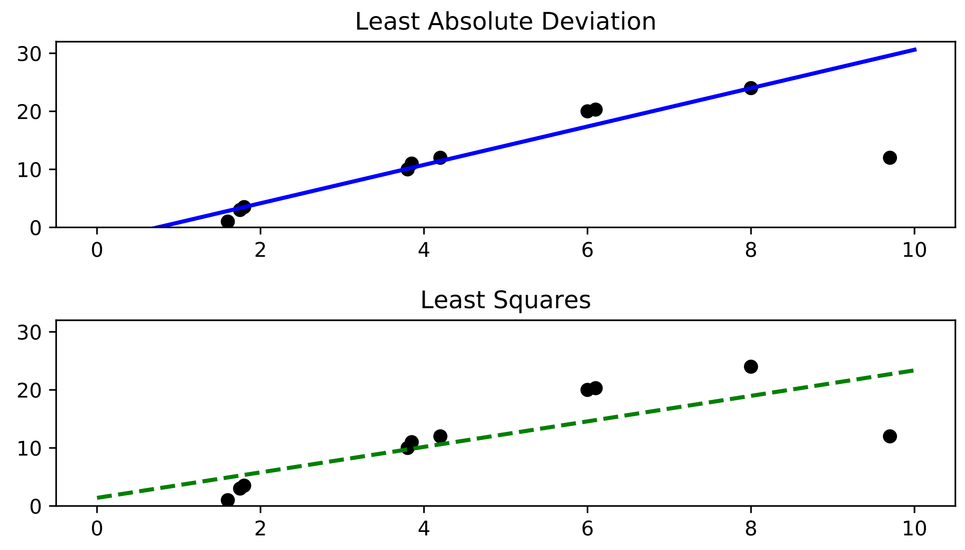

Figure 16.2: Fitted lines produced by least absolute deviations (top) and least squares (bottom). The presence of an outlier accounts for the stark difference between the two lines.

The u_i are implicitly constrained to be nonnegative.

Our linear program can now be stated as follows:

$$\begin{aligned} \text{minimize} \qquad & \sum_{i=1}^{m} u_i \\ \text{subject to} \qquad & u_i \geq \boldsymbol{\beta}^{\mathsf{T}}\mathbf{x}_i + b - y_i, \\ & u_i \geq y_i - \boldsymbol{\beta}^{\mathsf{T}}\mathbf{x}_i - b. \end{aligned}$$

Now for each inequality constraint, we bring all variables $(u_i, \boldsymbol{\beta}, b)$ to the left hand side and introduce a nonnegative slack variable to transform the constraint into an equality:

$$\begin{aligned} u_i - \boldsymbol{\beta}^{\mathsf{T}}\mathbf{x}_i - b - s_{2i-1} &= -y_i, \\ u_i + \boldsymbol{\beta}^{\mathsf{T}}\mathbf{x}_i + b - s_{2i} &= y_i, \\ s_{2i-1}, s_{2i} &\geq 0. \end{aligned}$$

Notice that the variables $\boldsymbol{\beta}, b$ are not assumed to be nonnegative, but in our interior point method, all variables are assumed to be nonnegative. We can fix this situation by writing these variables as the difference of nonnegative variables:

$$\begin{aligned} \boldsymbol{\beta} &= \boldsymbol{\beta}_1 - \boldsymbol{\beta}_2, \\ b &= b_1 - b_2, \\ \boldsymbol{\beta}_1, \boldsymbol{\beta}_2 &\succeq \mathbf{0}; b_1, b_2 \geq 0. \end{aligned}$$

Substituting these values into our constraints, we have the following system of constraints:

$$\begin{aligned} u_i - \boldsymbol{\beta}_1^{\mathsf{T}}\mathbf{x}_i + \boldsymbol{\beta}_2^{\mathsf{T}}\mathbf{x}_i - b_1 + b_2 - s_{2i-1} &= -y_i, \\ u_i + \boldsymbol{\beta}_1^{\mathsf{T}}\mathbf{x}_i - \boldsymbol{\beta}_2^{\mathsf{T}}\mathbf{x}_i + b_1 - b_2 - s_{2i} &= y_i, \\ \boldsymbol{\beta}_1, \boldsymbol{\beta}_2 &\succeq \mathbf{0}; u_i, b_1, b_2, s_{2i-1}, s_{2i} \geq 0. \end{aligned}$$

Writing $\mathbf{y} = (-y_1, y_1, -y_2, y_2, \ldots, -y_m, y_m)^\mathsf{T}$ and $\boldsymbol{\beta}_i = (\beta_{i,1}, \ldots, \beta_{i,n})^\mathsf{T}$ for $i = \{1, 2\}$, we can aggregate all of our variables into one vector as follows:

$$\mathbf{v} = (u_1, \ldots, u_m, \beta_{1,1}, \ldots, \beta_{1,n}, \beta_{2,1}, \ldots, \beta_{2,n}, b_1, b_2, s_1, \ldots, s_{2m})^\mathsf{T}.$$

Defining $\mathbf{c} = (1, 1, \ldots, 1, 0, \ldots, 0)^\mathsf{T}$ (where only the first m entries are equal to 1), we can write our objective function as

$$\sum_{i=1}^{m} u_i = \mathbf{c}^\mathsf{T}\mathbf{v}.$$

Hence, the final form of our linear program is:

$$\begin{aligned} \text{minimize} \quad & \mathbf{c}^\mathsf{T}\mathbf{v} \\ \text{subject to} \quad & A\mathbf{v} = \mathbf{y}, \\ & \mathbf{v} \succeq \mathbf{0}, \end{aligned}$$

where A is a matrix containing the coefficients of the constraints. Our constraints are now equalities, and the variables are all nonnegative, so we are ready to use our interior point method to obtain the solution.

LAD Example

Consider the following example. We start with an array `data`, each row of which consists of the values $y_i, x_{i,1}, \ldots, x_{i,n}$, where $\mathbf{x}_i = (x_{i,1}, x_{i,2}, \ldots, x_{i,n})^\mathsf{T}$. We will have $3m + 2(n + 1)$ variables in our linear program. Below, we initialize the vectors $\mathbf{c}$ and $\mathbf{y}$.

```
>>> m = data.shape[0]
>>> n = data.shape[1] - 1
>>> c = np.zeros(3*m + 2*(n + 1))
>>> c[:m] = 1
>>> y = np.empty(2*m)
>>> y[::2] = -data[:, 0]
>>> y[1::2] = data[:, 0]
>>> x = data[:, 1:]
```

The hardest part is initializing the constraint matrix correctly. It has $2m$ rows and $3m+2(n+1)$ columns. Try writing out the constraint matrix by hand for small m, n, and make sure you understand why the code below is correct.

```
>>> A = np.ones((2*m, 3*m + 2*(n + 1)))
>>> A[::2, :m] = np.eye(m)
>>> A[1::2, :m] = np.eye(m)
>>> A[::2, m:m+n] = -x
>>> A[1::2, m:m+n] = x
>>> A[::2, m+n:m+2*n] = x
>>> A[1::2, m+n:m+2*n] = -x
>>> A[::2, m+2*n] = -1
>>> A[1::2, m+2*n+1] = -1
>>> A[:, m+2*n+2:] = -np.eye(2*m, 2*m)
```

Now we can calculate the solution by calling our interior point function.

```
>>> sol = interiorPoint(A, y, c, niter=10)[0]
```

However, the variable `sol` holds the value for the vector

$$\mathbf{v} = (u_1, \ldots, u_m, \beta_{1,1}, \ldots, \beta_{1,n}, \beta_{2,1}, \ldots, \beta_{2,n}, b_1, b_2, s_1, \ldots, s_{2m+1})^{\mathsf{T}}.$$

We extract values of $\boldsymbol{\beta} = \boldsymbol{\beta}_1 - \boldsymbol{\beta}_2$ and $b = b_1 - b_2$ with the following code:

```
>>> beta = sol[m:m+n] - sol[m+n:m+2*n]
>>> b = sol[m+2*n] - sol[m+2*n+1]
```

Problem 5. The file `simdata.txt` contains two columns of data. The first gives the values of the response variables (y_i), and the second column gives the values of the explanatory variables $(\mathbf{x}_i)$. Find the least absolute deviations line for this data set, and plot it together with the data. Plot the least squares solution as well to compare the results.

```
>>> from scipy.stats import linregress
>>> slope, intercept = linregress(data[:,1], data[:,0])[:2]
>>> domain = np.linspace(0,10,200)
>>> plt.plot(domain, domain*slope + intercept)
```