CS5016: Computational Methods and Applications

Linear Systems and Interpolation

Albert Sunny

Department of Computer Science and Engineering
Indian Institute of Technology Palakkad

01 February, 2024

Linear systems

In applied science and engineering, one often faces equations of the following form

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

where $\boldsymbol{A}$ is a $n \times n$ square matrix whose elements are a_{ij}, $\boldsymbol{x}$ and $\boldsymbol{b}$ are column vectors of dimension n. Component-wise, the above equation can be written as

$$\sum_{j=1}^{n} a_{ij} x_j = b_i \quad \forall i \in \{1, 2, \ldots, n\}$$

$\boldsymbol{b}$ can also be interpreted as a linear combination of the columns of matrix $\boldsymbol{A}$ weighted by the vector $\boldsymbol{x}$.

An example

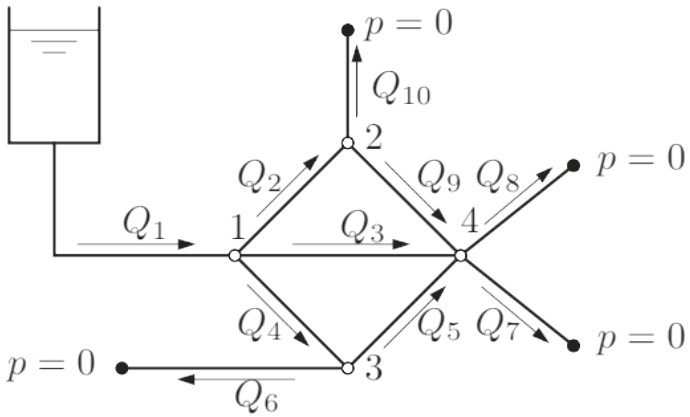

Figure: A pipe network[1].

Reservoir feeds water at a constant pressure of 10 *bar*. Let

$$Q_j = L_j \Delta p_j$$

where L_j is the length of pipe j and is given the product of indices of the nodes on either side of this pipe, and Δp_j is the pressure difference across the pipe.

Write a linear system of equations to compute the pressure at each node.

[1] "Scientific Computing with MATLAB and Octave", Alfio Quateroni and Fausto Saleri

Iterative solution method

An iterative method for the solution of the linear system results in a sequence of vectors $\{\boldsymbol{x}^{(k)}, k \geq 0\}$ of $\mathbb{R}^n$ that converges to the exact solution $\boldsymbol{x}^*$, that is

$$\lim_{k \to \infty} \boldsymbol{x}^{(k)} = \boldsymbol{x}^*,$$

for any given initial vector $\boldsymbol{x}^{(0)} \in \mathbb{R}^n$.

Constructing an Iterative Method

Split matrix $\boldsymbol{A}$, $\boldsymbol{A} = \boldsymbol{P} - (\boldsymbol{P} - \boldsymbol{A})$. $\boldsymbol{P}$ a suitable nonsingular matrix. Then

$$\boldsymbol{P}\boldsymbol{x}^* = \boldsymbol{b} - (\boldsymbol{A} - \boldsymbol{P})\boldsymbol{x}^* \tag{1}$$

Correspondingly, for $k \geq 0$ we can define the following iterative method:

$$\boldsymbol{P}\boldsymbol{x}^{(k+1)} = \boldsymbol{b} - (\boldsymbol{A} - \boldsymbol{P})\boldsymbol{x}^{(k)} \tag{2}$$

(2) - (1) gives us

$$\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^* = (\boldsymbol{I} - \boldsymbol{P}^{-1}\boldsymbol{A})(\boldsymbol{x}^{(k)} - \boldsymbol{x}^*) \tag{3}$$

Convergence

Let $\boldsymbol{e}^{(k)} = \boldsymbol{x}^{(k)} - \boldsymbol{x}^*$ denote the error at step k.

If $(\boldsymbol{I} - \boldsymbol{P}^{-1}\boldsymbol{A})$ is symmetric and positive definite, we have

$$\|\boldsymbol{e}^{(k+1)}\|_2 = \|(\boldsymbol{I} - \boldsymbol{P}^{-1}\boldsymbol{A})\boldsymbol{e}^{(k)}\|_2 \leq \rho(\boldsymbol{I} - \boldsymbol{P}^{-1}\boldsymbol{A})\|\boldsymbol{e}^{(k)}\|_2$$

where $\rho(\cdot)$ is known as the *spectral radius* (maximum modulus of eigenvalues). If $\rho(\cdot) < 1$, there is convergence.

The Jacobi method

If the diagonal entries of $\boldsymbol{A}$ are nonzero, we can set $\boldsymbol{P} = \boldsymbol{D}$, where $\boldsymbol{D}$ is the diagonal matrix containing the diagonal entries of $\boldsymbol{A}$. Then, we get the following iteration

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1, j \neq i}^{n} a_{ij} x_j^{(k)} \right) \qquad \forall i \in \{1, 2, \ldots, n\}$$

Proposition

If the matrix $\boldsymbol{A}$ is strictly diagonally dominant by row, then the Jacobi method converges.

Try proving the above proposition.

The Gauss-Siedel method

Faster convergence could be (hopefully) achieved if the new (k+1) components already available are used

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) \quad \forall i \in \{1, 2, \ldots, n\}$$

Caution

There are no general results stating that the Gauss-Seidel method converges faster than Jacobi's

Python's numpy.linalg module

The NumPy linear algebra functions rely on BLAS and LAPACK to provide efficient low level implementations of standard linear algebra algorithms.

To know more, visit
`https://numpy.org/doc/stable/reference/routines.linalg.html`

Interpolation

In several applications we may only know value of a function f at some given points $\{(x_i, y_i), i = 0, 1, 2, \ldots, n\}$. How do we determine f?

In such a situation it is natural to figure out an approximate function $\tilde{f}$ that satisfies the following

$$\tilde{f}(x_i) = y_i \quad \forall i \in \{0, 1, 2, \ldots, n\}$$

Can you figure out a cubic function that passes through the points $(0, 1)$, $(1, 4)$, $(-1, 0)$ and $(2, 15)$? How many such such cubic functions exist?

A possible way!!!

Assume

$$f(x) = a + bx + cx^2 + dx^3$$

Then we get the following linear system

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 2 & 4 & 8 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \\ 0 \\ 15 \end{bmatrix}$$

Solving the above linear system will give us coefficients of the desired cubic polynomial.

What is the complexity of the above method?

Different kinds of interpolation

- *polynomial interpolation*

$$\tilde{f}(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

- *trigonometric interpolation*

$$\tilde{f}(x) = a_{-M} e^{-iMx} + \cdots + a_0 + \cdots + a_M e^{iMx}$$

- *rational interpolation*

$$\tilde{f}(x) = \frac{a_0 + a_1 x + \cdots + a_k x^k}{b_0 + b_1 x + \cdots + b_n x^n}$$

Lagrangian polynomial interpolation

For $j \in \{0, 1, \ldots, n\}$, define

$$\psi_j(x) = \prod_{i=0, i \neq j}^{n} \frac{x - x_i}{x_j - x_i}$$

Note that

$$\psi_j(x_k) = \begin{cases} \prod_{i=0, i \neq j}^{n} \frac{x_j - x_i}{x_j - x_i} = 1 & \text{if } k = j \\ 0 & \text{otherwise} \end{cases}$$

Then, the required approximation is

$$\tilde{f}(x) = \sum_{j=0}^{n} y_j \psi_j(x)$$

Coefficients of the above polynomial can be computed in $O(n^2)$ time.

An example

Consider a function that passes through the points $(0, 1)$, $(1, 4)$, $(-1, 0)$ and $(2, 15)$.

$$\psi_1(x) = \frac{x^3 - 2x^2 - x + 2}{2}$$

$$\psi_2(x) = \frac{-x^3 + x^2 + 2x}{2}$$

$$\psi_3(x) = \frac{-x^3 + 3x^2 - 2x}{6}$$

$$\psi_4(x) = \frac{x^3 - x}{6}$$

and we have

$$\begin{aligned}\tilde{f}(x) &= \psi_1(x) + 4\psi_2(x) + 15\psi_4(x) \\ &= x^3 + x^2 + x + 1\end{aligned}$$

Python's scipy.interpolate module

Contains spline functions and classes, 1-D and multidimensional (univariate and multivariate) interpolation classes, Lagrange and Taylor polynomial interpolators, and wrappers for FITPACK and DFITPACK functions.

To know more, visit
`https://docs.scipy.org/doc/scipy/reference/interpolate.html`

Thank You