Lower Triangular solve

In [1]:
```
import numpy as np
```

In [2]:
```
def lower_triangular_solve(A, b):
    """
    Solve the system  A x = b  where A is assumed to be lower triangular,
    i.e. A(i,j) = 0 for j > i, and the diagonal is assumed to be nonzero,
    i.e. A(i,i) != 0.

    The code checks that A is lower triangular and converts A and b to
    double precision before computing.

    ARGUMENTS:  A   lower triangular n x n array
                b   right hand side column n-vector

    RETURNS:    x   column n-vector solution
    """
    # we should take care to ensure that arrays are stored with the corre
    A = A.astype(np.float64)
    b = b.astype(np.float64)
    #check that the sizes of A and b match and A is square
    if A.shape[0] != A.shape[1] or A.shape[0] != len(b):
        raise ValueError("invalid input shapes")

    # check that A is lower triangular
    for i in range(A.shape[0]):
        #check that A does not have any zero diagonal elements
        if np.isclose(A[i,i], 0.0):
            raise ValueError("A contains zero diagonal element(s)")
        for j in range(i+1, A.shape[0]):
            if not np.isclose(A[i,j], 0.0):
                raise ValueError("A is not lower triangular")
    #initilaise result array
    x = np.empty_like(b)
    #perform forward substitution
    x[0] = b[0] / A[0,0]
    for i in range(1,A.shape[0]):
        x[i] = b[i] /A[i,i]
        for j in range(i):
            x[i] = x[i] - x[j] * A[i,j] / A[i,i]
    return x
```

A lower triangular matrix is a matrix where all the entries above the diagonal are zero. To solve the equation Ax = b for x, where A is a lower triangular matrix and b is a vector, we can solve using forwards substition. For each value in b, b[i], the corresponding value in x,

$$x[i] = b[i]/A[i,i] - sum(j = 0 \Rightarrow (i-1), A[i,j] * x[j]/A[i,i])$$

Upper Triangular Solve

In [3]:
```python
def upper_triangular_solve(A, b):
    """
    Solve the system  A x = b  where A is assumed to be upper triangular,
    i.e. A(i,j) = 0 for j < i, and the diagonal is assumed to be nonzero,
    i.e. A(i,i) != 0.

    The code checks that A is upper triangular and converts A and b to
    double precision before computing.

    ARGUMENTS:  A   upper triangular n x n array
                b   right hand side column n-vector

    RETURNS:    x   column n-vector solution
    """
    # we should take care to ensure that arrays are stored with the corre
    A = A.astype(np.float64)
    b = b.astype(np.float64)
    #check that the sizes of A and b match and A is square
    if A.shape[0] != A.shape[1] or A.shape[0] != len(b):
        raise ValueError("invalid input shapes")

    #check that A is upper triangular
    for i in range(A.shape[0]):
        #check there are no zero diagonal entries
        if np.isclose(A[i,i], 0.0):
            raise ValueError("zero diagonal entry found")
        for j in range(0, i):
            if not np.isclose(A[i,j], 0.0):
                raise ValueError("not upper triangular")

    n = A.shape[0]
    #initialise output array
    x = np.empty_like(b)

    #perform backward substitution
    x[n-1] = b[n-1] / A[n-1,n-1]
    for i in range(2, n+1):
        x[n-i] = b[n-i] / A[n-i,n-i]
        for j in range(n-i+1, n):
            x[n-i] = x[n-i] - A[n-i,j] * x[j] / A[n-i,n-i]
    return x
```

An upper triangular matrix is a matrix where all the entries below the diagonal are zero. To solve the equation Ax = b for x, where A is n upper triangular matrix and b is a vector, we can solve using backwards substitution. For each value in b, b[i], the correponding value in x,

$$x[i] = (b[i] - sum(j = i+1 \Rightarrow n, A[i][j] * x[j]))/A[i][i]$$

Triangular solve input checking

for triangular solves, we need to make sure that our matrix is triangular, and therefore square, and has no zero diagonal entries. We also need to check that the width of the matrix is the same as the length of the resultant vector we also need to make sure that our matrices and vectors are in np.double form

Guassian elimination

In [4]:
```
def gaussian_elimination(A, b, verbose=False):
    """
    Reduce the system  A x = b  to upper triangular form, assuming that
    the diagonal is nonzero, i.e. A(i,i) != 0.

    Before computing A and b are converted to double precision.

    ARGUMENTS:  A   n x n matrix
                b   right hand side column n-vector

                verbose  (optional) if true print elimination steps

    RETURNS:    A   upper triangular n x n matrix
                b   modified column n-vector
    """
    # we should take care to ensure that arrays are stored with the corre
    A = A.astype(np.float64)
    b = b.astype(np.float64)

    #check that A is square and the lengths  b = A[0]
    if A.shape[0] != A.shape[1] or A.shape[0] != len(b):
        raise ValueError("Invalid input shapes")

    #perform forward elimination
    for i in range(A.shape[0]):
        #check for zero on the diagonal
        if np.isclose(A[i,i], 0.0):
            raise ValueError("zero diagonal entry found")
        # row j = row j - row i * A[j,i] / A[i,i]
        for j in range(i+1, A.shape[0]):
            factor = A[j,i] / A[i,i]
            for k in range(A.shape[0]):
                A[j,k] = A[j,k] - A[i,k] * factor
            b[j] = b[j] - b[i] * factor
    #return A, b
    return upper_triangular_solve(A,b)
```

Guassian elimination works by subtracting each row, multiplied by a factor, from each row below it to create a triangular matrix

LU factorisation

```
In [5]: def LUFactorise(A):
            L = np.zeros_like(A)
            U = np.zeros_like(A)
            for j in range(A.shape[0]):
                #set L diagonal to 1
                L[j,j] = 1.0
                #calculate U values
                for i in range(j+1):
                    U[i,j] = A[i,j] - sum(U[k,j] *L[i,k] for k in range(i))
                #calculate L values
                for i in range(j+1, n):
                    L[i,j] = (A[i,j] - sum(U[k,j] * L[i,k] for k in range(j)))/U[
            return L, U
```

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} u_{kj} l_{ik}$$

$$l_{ij} = \frac{1}{u_{jj}} (a_{ij} - \sum_{k=1}^{j-1} u_{kj} l_{ik})$$

Iterative methods

Used to solve systems of equations in faster time, losing precision. Guass Seidel is literally the same as Jacobi but it uses a more recent version of the other xs when iterating

In [6]:
```python
def Jacobi_iteration(A, b, max_iteration, x0 = None, verbose = False):
    # we should take care to ensure that arrays are stored with the corre
    A = A.astype(np.float64)
    b = b.astype(np.float64)

    # check sizes of A and b match appropriately
    nb=len(b)
    n, m = A.shape
    if n != m:
        raise ValueError(f'A is not a square matrix! {A.shape=}')
    if n != nb:
        raise ValueError(f'shapes of A and b do not match! {A.shape=} {b.

    # check diagonal is non zero
    for i in range(n):
        if np.isclose(A[i, i], 0):
            raise ValueError(f'A[{i}, {i}] is zero')

    # construct iteration matrices
    P=np.zeros([n,n])    # matrix P = D^{-1}(L+U)
    p=np.zeros(n)        # vector p = D^{-1} b
    for i in range(n):
        p[i]=b[i]/A[i,i]
        for j in range(n):
             P[i,j] = A[i,j]/A[i,i]
        P[i,i] = 0

    #create a new array to store the results, initialised as zero
    if x0 is None:
        x = np.zeros_like(b)
    else:
        x = x0.copy()

    # perform iteration x <- p - P * x
    for i in range(max_iteration):
        xnew = np.empty_like(x)
        for i in range(n):
            xnew[i] = p[i]
            for j in range(n):
                xnew[i] -= P[i, j] * x[j]
        x = xnew.copy()
        if verbose:
            print("iteration ", i+1)
            print(x)

    return x
```

In [7]:
```
def Gauss_Seidel_iteration(A, b, max_iteration, x0 = None, verbose = Fals
    # we should take care to ensure that arrays are stored with the corre
    A = A.astype(np.float64)
    b = b.astype(np.float64)

    # check sizes of A and b match appropriately
    nb=len(b)
    n, m = A.shape
    if n != m:
        raise ValueError(f'A is not a square matrix! {A.shape=}')
    if n != nb:
        raise ValueError(f'shapes of A and b do not match! {A.shape=} {b.

    for i in range(n):
        if np.isclose(A[i, i], 0):
            raise ValueError(f'A[{i}, {i}] is zero')

    # do not construct iteration matrices explicitly
    LD = np.zeros_like(A)
    U = np.zeros_like(A)
    for i in range(n):
        for j in range(n):
            if i < j:
                U[i, j] = A[i, j]
            else:
                LD[i, j] = A[i, j]

    # p = (L + D)^{-1} b --> found by solving triangular system
    # (L + D) p = b
    p = lower_triangular_solve(LD, b)

    #create a new array to store the results, initialised as zero
    if x0 is None:
        x = np.zeros_like(b)
    else:
        x = x0.copy()

    # perform iteration x <- p - P * x
    # (L+D)(xnew - p) = U*x
    Ux = np.empty_like(x)
    for i in range(max_iteration):
        for i in range(n):
            Ux[i] = 0.0
            for j in range(i+1, n):
                Ux[i] += U[i, j] * x[j]
        Px = lower_triangular_solve(LD, Ux)
        x = p - Px
        if verbose:
            print("iteration ", i+1)
            print(x)

    return x
```

Sparse multiplication

We like sparse matrices because we can operate on them in O(n) time because there is only alpha n amount of data in them, where alpha is a constant not related to n.

```
In [9]: #z is the outcome of the multiplication
        #I_row is the array of rows values belong to
        #I_col is the array of columns values belong to
        #A_real is the array of values
        #y is the multiplier
        for k in range(len(A_real)):
            z[I_row[k]] = z[I_row[k]] + A_real[k] * y[I_col[k]]
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call las
/tmp/ipykernel_21798/2359271269.py in <module>
      4 #A_real is the array of values
      5 #y is the multiplier
----> 6 for k in range(len(A_real)):
      7     z[I_row[k]] = z[I_row[k]] + A_real[k] * y[I_col[k]]

NameError: name 'A_real' is not defined
```

Guassian Elimination with pivoting

When we are guassian eliminating, we can end up with 1/ a very small number. This results in a very large number, and gives us big problems with precision. In order to do this, we pivot our matrix such that we are always dividing by the biggest number possible. Pivoting is literally just swapping rows in both A and b. This gives as the least error, however it is still possible to encounter it if even the biggest number is still close to zero.

```
In [10]: def Gaussian_elimination_pivoting(A, b, verbose=False):
             # To ensure that arrays are stored in double precision.
             A = A.astype(np.float64)
             b = b.astype(np.float64)

             # size of solution vector / the square matrix A
             n=len(b) # or   n, n = A.shape

             # check sizes of A and b match appropriately
             nb=len(b)
             n, m = A.shape
             if n != m:
                 raise ValueError(f'A is not a square matrix! {A.shape=}')
             if n != nb:
                 raise ValueError(f'shapes of A and b do not match! {A.shape=} {b.

             if verbose:
                 print('starting system\n', A, b)

             # perform forward elimination
             for i in range(n):
                 # find the index of the maximal value in column i on or below
                 # the diagonal of A
                 maximum = abs(A[i,i])
                 max_index = i
                 for j in range(i+1,n):
                     if abs(A[j,i]) > maximum :
                         maximum = abs(A[j,i])
                         max_index = j


                 # swap two max_indexs: i and max_index[i]
                 temp = b[i]
                 b[i] = b[max_index]
                 b[max_index] = temp
                 for j in range(n):
                     temp = A[i,j]
                     A[i,j] = A[max_index,j]
                     A[max_index,j] = temp


                 # check diagonal
                 if np.isclose(A[i, i], 0.0):
                     raise ValueError(f'A has zero on diagonal! A[{i}, {i}] = 0')

                 # row j <- row j - (a_{ji} / a_{ii}) row i
                 for j in range(i+1, n):
                     if verbose:
                         print(f'row {j} <- row {j} - {A[j, i] / A[i, i]} row {i}'
                     factor = A[j, i] / A[i, i]
                     for k in range(0, n):
                         A[j, k] = A[j, k] - factor * A[i, k]
                     b[j] = b[j] - factor * b[i]

             return upper_triangular_solve(A, b)
```

Euler's method

Euler's method is used to solve systems with differential equations. It is not exact, and smaller step size = better

In [11]:
```
def Euler_method(t0, d0, dt, n, f):
    d = np.zeros(n+1)
    d[0] = d0

    t = np.zeros(n+1)
    for i in range(n+1):
        t[i] = t0 + i*dt

    for i in range(1, n+1):
        d[i] = d[i-1] + dt * f(t[i-1], d[i-1])

    return t, d
```

Midpoint method

The midpoint method is an improvement in eulers method which provides a lower error for the same number of calculations (although there are double the number of calculations per step)

In [12]:
```
def midpoint_method(t0, d0, dt, n, f):
    d = np.zeros(n+1)
    d[0] = d0

    t = np.zeros(n+1)
    for i in range(n+1):
        t[i] = t0 + i*dt

    for i in range(1, n+1):
        d_half = d[i-1] + 0.5 * dt * f(t[i-1], d[i-1])
        d[i] = d[i-1] + dt * f((t[i-1]+t[i])/2.0, d_half)

    return t, d
```

Root finding

Newton's method and the bisection method are both iterative methods to find the roots of non-linear equations. Newton's method converges faster The bisection method will always find the root, but Newton's method may not

In [13]:
```
def newton(f, df, x0, tol, verbose = False):
    x = x0
    y = f(x)
    it = 0
    while abs(y) > tol:   # iterate until less than or eq tol
        x = x - y / df(x)  # apply one Newton iteration
        y = f(x)           # reevaluate f at new estimate
        it = it + 1
        if verbose:
            print("iteration ", it)
            print(x)

    return x, it


def bisection(f, x0, x1, tol, verbose = False):
    it = 0
    x = (x0 + x1)/2.0
    while abs(f(x)) > tol:
        it = it +1
        x = (x0 + x1)/2.0
        if abs(x) < 1.e-6: return x
        if f(x)*f(x0) < 0:
            x1 = x
        else:
            x0 = x
        if verbose:
            print("iteration ", it)
            print(x)

    return x, it
```

In [15]:
```
def secant(f, x0, x1, tol):
    x = x1
    it = 0
    while abs(f(x)) > tol:   # iterate until less than or eq tol
        x = x - f(x1) *(x1-x0) / (f(x1) - f(x0))  # apply one Newton iter
        x0 = x1
        x1 = x
        it = it + 1

    return x, it
```

Least squares fitting

When we have too many equations for the number of unknowns, we cannot solve them exactly.

We still construct our A and b the same.

We then construct

$$A^T A x = A^T A b$$

This gives us a system of equations which can be solved with guassian elimination for x.

Something on the course I could use again

Guass-Seidel method can be useful. I may have to use it in the future in computer graphics problems. It is useful for this application as it solves the problem efficiently, with the only drawback that it is not an exact solution, which is generally a fine compromise to make in computer graphics.

```
In [ ]:
```