Important Numerical Programs

Ajit Rajwade

Contents

- Gradient Descent
- Solutions to linear simultaneous equations: Gaussian elimination (also called LU decomposition or LU factorization)

Gradient Descent to Minimize a Function

Gradient Descent

- Consider a function `f(x,y)` in variables `x,y`.
- Suppose we want to find the value of `(x,y)` for which `f(x,y)` is **minimum**. This is called **minimization of a function**.
- Assuming that `f` is differentiable (almost) everywhere, a popular method for this is called **gradient descent** (also called **steepest descent**).
- It is very **popular** in machine learning, neural networks, image processing, artificial intelligence, computer graphics and various other allied fields.
- It is nice that you are getting to study about it in your first semester :-)
- Uses the following fact: Given a point `(x0,y0)`, the function `f(x,y)` decreases fastest in the direction of the negative gradient, i.e. – `(fx(x0,y0), fy(x0,y0))`.
- Here `fx(x0,y0)` stands for the partial derivative of `f(x,y)` at `(x0,y0)` w.r.t. `x`, assuming `y` is a constant.
- Likewise `fy(x0,y0)` stands for the partial derivative of `f(x,y)` at `(x0,y0)` w.r.t. `y`, assuming `x` is a constant.

Gradient Descent

- The gradient descent algorithm takes small steps in the direction of the negative in the following manner:

$$(x,y) = (x,y) - \alpha\ (fx(x,y), fy(x,y))$$

- To clarify, we have: $x_{i+1} = x_i - \alpha fx(x_i, y_i)$; $y_{i+1} = y_i - \alpha fy(x_i, y_i)$ where x_i, y_i are estimates of x and y in the i-th iteration.
- Here $\alpha > 0$ is called the **learning rate** or the **step size** of gradient descent.
- The above update is executed for several iterations until you reach a point where the values of $fx(x,y)$ and $fy(x,y)$ are both zero or very close to zero.
- At that point you have reached close to a minimum of f.
- Note that this is a local minimum, and it will satisfy the property that $fx(x,y) = fy(x,y) = 0$.
- We have explained everything for two-dimensional functions, but everything is applicable to functions in any dimension including in one dimension.

```
main_program{
// function f(x) = (x-3)^2+5;
double epsilon = 0.001; // desired precision
double x, fx, dfx; // to ensure that the while loop is entered
double stepsize;
cout << "enter the initial guess: "; cin >> x;

fx = (x-3)*(x-3)+5; dfx = 2*(x-3);
cout << x << " " << fx << " " << dfx << endl;
stepsize = 0.05;
while (fabs(dfx) > epsilon)
{
    x = x - stepsize * dfx; // the main descent step
    fx = (x-3)*(x-3)+5; // function value
    dfx = 2*(x-3); // derivative value
    cout << x << " " << fx << " " << dfx << endl;
    wait (0.5);
}
cout << "The minimum is at " << x;
}
```

Gradient descent

- In the earlier program, a fixed step-size of 0.05 was used.
- This may not always be optimal – **luckily** it worked well for the previous problem!
- In practice, an **adaptive step-size** strategy is chosen.
- **Strategy 1: Line Search**
  - You choose a range of step-sizes from a minimum (stepsize_min) to a maximum value (stepsize_max)
  - For every value (called it gamma) in this range, in small steps (of say 0.001), determine f(x1gamma,y1gamma) where x1gamma = x - gamma fx(x,y) and y1gamma = y1 - gamma fy(x,y).
  - Choose the gamma value that gave the **least** value of f(x1gamma,y1gamma).

Gradient descent

- **Strategy 2: Backtracking Search**
  - You set step-size to some large value, eg 2, and check whether the descent step actually decreased the function value.
  - If it decreased the value of the function, you continue with the updated value of x in further iterations.
  - If it increased the function value, you ignore the updated value of x. Now, you decrease the step-size and again repeat the earlier two steps.
  - If despite reducing the step-size to some lower bound such as 1e-4 the function does not reduce in value, it means you had already reached a minimum.
- You will implement one of these strategies yourself in one of the (ungraded) lab sessions!
- Once you have done that, you have effectively done assignment#1 of NN101 (Neural Networks 101) :-) as neural networks primarily use some variant of gradient descent.
- **Note: in both strategies, the step-size will generally change across the iterations!**

Solutions to Linear Simultaneous Equations via Gaussian Elimination (also called LU decomposition)

Simultaneous Linear Equations

- We have studied linear simultaneous equations in high school, mostly in 2-3 variables.
- For example: x + y + z = 10, 2x - 3y = 4; 3x + 4y + 5z = 6;
- In engineering, there are many areas where such equations arise in thousands of variables.
- We need computerized methods for solving such equations.
- A matrix representation for such equations is very convenient.
- This takes the form **Ax** = **b** where **A** is a *n* x *n* known coefficient matrix, **x** is *n* x 1 vector of unknown values and **b** is a *n* x 1 vector of known coefficients.
- In the above example, this takes the form:

$$\begin{pmatrix} 1 & 1 & 1 \\ 2 & -3 & 0 \\ 3 & 4 & 5 \end{pmatrix} \boldsymbol{x} = \begin{pmatrix} 10 \\ 4 \\ 6 \end{pmatrix}$$

Upper triangular systems

- A square matrix is one in which the number of rows and columns are equal.
- A square matrix is said to be an upper triangular matrix if all elements below the diagonal are zero.
- Example in a linear system **Ux** = **b**, consider:

$$U = \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & \cdots & u_{1,n} \\ & u_{2,2} & u_{2,3} & \cdots & u_{2,n} \\ & & \ddots & \ddots & \vdots \\ & & & \ddots & u_{n-1,n} \\ 0 & & & & u_{n,n} \end{bmatrix}$$

Upper triangular systems

- Such systems are easy to solve using a method called back-substitution.
- First solve $x_n = b_n/u_{nn}$ (assuming unn is not equal to 0).
- For i=n-1 to 1, we have:

$$x_i = \frac{b_i - \sum_{j=i+1}^{n} u_{ij} x_j}{u_{ii}}$$

$$U = \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & \cdots & u_{1,n} \\ & u_{2,2} & u_{2,3} & \cdots & u_{2,n} \\ & & \ddots & \ddots & \vdots \\ & & & \ddots & u_{n-1,n} \\ 0 & & & & u_{n,n} \end{bmatrix}$$

Code for back-substitution

```
void backsub_LU (float A[][MAX], float *b, float *x, int n){
int i,j;
for(i=n-1; i>= 0; i--){
if (A[i][i] == 0) { cout << "matrix is singular"; return;}
    x[i] = b[i];
     for(j=i+1;j<=n-1;j++){
             x[i] = x[i] - A[i][j]*x[j];
     }
     x[i] = x[i]/A[i][i];
}
}
```

Time complexity: O(n^2)
Space complexity: O(n), due to space required to store x.

```
int main(){
        int n,i,j;
        float A[MAX][MAX],x[MAX],b[MAX];

        cout << "enter n";
        cin >> n;

        cout << endl << "enter the values in A" << endl;
        for(i=0;i<n;i++){
                for(j=0;j<n;j++){
                        cout << "enter value at (" << i << "," << j << ")";
                        cin >> A[i][j];
                }
        }
        cout << endl << "enter the values of b";
        for(i=0;i<n;i++) cin >> b[i];
        backsub_LU (A,b,x,n);

        cout << endl;
        for(i=0;i<n;i++) cout << x[i] << " ";
}
```

```
enter n: 3

enter the values in A
enter value at (0,0)1
enter value at (0,1)2
enter value at (0,2)3
enter value at (1,0)0
enter value at (1,1)1
enter value at (1,2)1
enter value at (2,0)0
enter value at (2,1)0
enter value at (2,2)3

enter the values of b: 4
5
6

x: -8 3 2
```

```
enter n: 3

enter the values in A
enter value at (0,0)1
enter value at (0,1)2
enter value at (0,2)3
enter value at (1,0)0
enter value at (1,1)2
enter value at (1,2)1
enter value at (2,0)0
enter value at (2,1)0
enter value at (2,2)3

enter the values of b: 4
5
6

x: -5 1.5 2
```

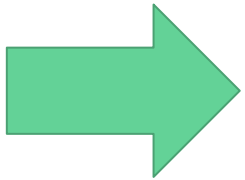
Gaussian Elimination

- Gaussian elimination is the process of converting any linear system Ax = b into an equivalent system of the form Ux = b’ where U is an upper triangular matrix.
- Also b’ is a suitably modified version of b.
- The system Ux = b’ can now be solved using back-substition.
- For example, consider the system of equations below:

$x_1 + 2x_2 + 2x_3 = 3$

$4x_1 + 4x_2 + 2x_3 = 6$

$4x_1 + 6x_2 + 4x_3 = 10$



$$\begin{pmatrix} 1 & 2 & 2 \\ 4 & 4 & 2 \\ 4 & 6 & 4 \end{pmatrix} \boldsymbol{x} = \begin{pmatrix} 3 \\ 6 \\ 10 \end{pmatrix}$$

We will seek to convert this system into one with an upper triangular matrix on the left side

Gaussian Elimination

$$\begin{pmatrix} 1 & 2 & 2 \\ 4 & 4 & 2 \\ 4 & 6 & 4 \end{pmatrix} \boldsymbol{x} = \begin{pmatrix} 3 \\ 6 \\ 10 \end{pmatrix}$$

Perform an operation: R2 - 4 times R1, and b2-4 times b1. The factor 4 is because A[1][0]/A[0][0] = 4. This operation will make A[1][0] = 0. Here A[0][0] is called the pivot.

$$\begin{pmatrix} 1 & 2 & 2 \\ 0 & -4 & -6 \\ 4 & 6 & 4 \end{pmatrix} \boldsymbol{x} = \begin{pmatrix} 3 \\ -6 \\ 10 \end{pmatrix}$$

Perform an operation: R3 - 4 times R1, and b3 - 4 times b1. The factor 4 is because A[2][0]/A[0][0] = 4. This operation will make A[2][0] = 0. Here A[0][0] is the pivot.

$$\begin{pmatrix} 1 & 2 & 2 \\ 0 & -4 & -6 \\ 0 & -2 & -4 \end{pmatrix} \boldsymbol{x} = \begin{pmatrix} 3 \\ -6 \\ -2 \end{pmatrix}$$

Perform an operation: R3 - 0.5 times R2, and b3 - 0.5 times b2. The factor 0.5 is because A[2][1]/A[1][1] = 0.5. This operation will make A[2][0] = 0. Here A[1][1] is the pivot.

Gaussian Elimination

$$\begin{pmatrix} 1 & 2 & 2 \\ 0 & -4 & -6 \\ 0 & 0 & -1 \end{pmatrix} \boldsymbol{x} = \begin{pmatrix} 3 \\ -6 \\ 1 \end{pmatrix}$$

We now have an upper triangular system which we can solve by back-substitution which produces x3 = -1, x2 = 3, x1= -1.

We worked out Gaussian elimination for a 3 x 3 system, but the same logic can be extended to a 4 x 4 system or any n x n system. In case of a n x n system, the pivots will be A[0][0], A[1][1], ..., A[n-1][n-1].

When the pivot is A[i][i], the following operation needs to be performed on all rows indexed by j from i+1 till n-1:
For every k-th element (k from 0 to n-1), do the operation: A[j][k] = A[j][k] - ratio*A[i][k]
where ratio = A[j][i]/A[i][i]

Lastly b[j] = b[j] - ratio*b[i].

Repeat these operations for all pivots.

```
void GaussElim (float A[][MAX], float *b, int n){
        int i,j,k;
        float ratio;

        for(i=0;i<n-1;i++){
                if (A[i][i] == 0) {cout << "mathematical error";
return;}
                for(j=i+1;j<n;j++){
                        ratio = A[j][i]/A[i][i];
                        for(k=0;k<n;k++){
                                A[j][k] = A[j][k]-ratio*A[i][k];
                        }
                        b[j] = b[j]-ratio*b[i];
                }
        }
}
```

Time complexity: O(n^3)
Space complexity: O(1)