CODE DOCUMENTATION FOR NEURON MODEL:

1. Variable Description:

1.1 Transmitter *part*:

The variables used in transmitter part of the code are described in the following table:

1.1.1 Data Members for TRANSMITTER part:

Sl. No.	Variable Name		Variable Data type	Variable Description
	TRANSMITTER			
1	x_distance		double	A variable that stores the x-distance from the transmitter
2	y_distance		double	A variable that stores the y-distance from the transmitter
3	Diffusion_coefficient		double	A variable that stores the value of diffusion coefficient of the diffusion process
4	arrival_rate		int	A variable that stores the calcium generation rate
5	time_interval		double	A variable that stores the value of time instant
6	number_of_elements_generated		int	A variable that stores the number of elements generated in that time interval
	POISSON GENERATION			
1	L		double	floating point variable that stores the exponential inter-arrival time depending upon the generation rate (λ)
2	p		double	the incremental time value which increments in each loop execution until its value is less than L.
3	u		double	random value between [0,1] generated in each loop execution.

4	k		int	counts the number of ions/molecules produced during the time L.
	MAIN FUNCTION			
1	number_of_calcium_generated		int	A variable that stores the number of calcium ions generated in the time instant
2	number_of_BDNF_generated		int	A variable that stores the number of BDNF molecules generated in the time instant

1.1.2 Member functions of TRANSMITTER part:

The member function of the transmitter part takes in two variable "x_distance" and "initial_conc", generates the number of calcium ions generated in the time interval which will be used in the transport part (gates) and for metric calculation.

Function declaration for active transport is:

int transmitter(int);

The member function of the poisson generation takes in generation rate "lambda" and returns the number of poisson distributed random molecules/ions generated in the time interval.

Function declaration for diffusion transport is:

int poisson_generation(double);

1.2 Transport part:

The variables used in transport part of the code are described in the following table:

1.2.1 Data Members for TRANSPORT part:

Sl. No.	Variable Name	Variable Data type	Variable Description
ACTIVE TRANSPORT			
1	start	nodedata	A queue of type nodedata used to store the number of BDNF molecules in MTs
2	temp	nodedata	A pointer to starting of queue used to perform some manipulations on the queue
3	gap_index	int	An integer value for storing the index of the inter-tubular gaps
4	i	int	A loop counter variable
5	num_calcium	int	An integer variable that stores the number of calcium ions generated in each time interval
6	num_BDNF	int	An integer variable that stores the number of BDNF molecules generated in each time interval
7	arr_pkt	int	A loop counter that runs for number of BDNF molecules generated
8	sw_car	int	A loop counter that runs for number of cargoes in the MT segment.
9	sw_pkt	int	A loop counter that runs for number of BDNF molecules served in the MT.
10	src	int	An integer variable that stores the index of the source MT
11	des	int	An integer variable that stores the index of the destination MT
12	src_n	int	A loop variable that runs for number of MT segments in the MT
13	b	int	An integer array for storing the buffer value of the MT queue.
14	count	int	A counter variable for counting the number of BDNF molecules in the queues.
15	count2	int	A counter variable for counting the number of BDNF molecules in the queues.
16	tot_arv_calcium	int	An integer variable that stores the number of calcium ions generated by the transmitter.

17	tot_arv_BDNF	int	An integer variable that stores the number of BDNF molecules generated in the transmitter
18	tot_lost	int	A variable that stores the number of BDNF molecules lost in the queues
19	tot_dep	int	A variable that stores the number of BDNF molecules served by the queues
20	value	int	A variable that indicates the buffer of the queue is not full
21	Level	int	A loop counter that runs for number of levels in the queue.
22	mean_arr_rate	double	A variable that stores the average arrival rates of BDNF molecules in the transmitter region.
23	mean_srv_rate	double	A variable that stores the average service rate of the BDNF molecules in the MT queue.
24	mean_arr_rate_ca	double	A variable that stores the average arrival rates of Ca^{2+} ions in the transmitter region.
25	mean_srv_rate_ca	double	A variable that stores the average service rate of Ca^{2+} ions in the MT queue.
26	l_mt	double	A variable that stores the length of the microtubule
27	v_cargo	double	A variable that stores the velocity of the cargo
28	mean_sw_rate	double	A variable that stores the net service rate of the BDNF molecules taking into consideration the effect of Ca^{2+} ions on BDNF transport.
29	pr	double	A Double array of three elements for storing the value of fraction of BDNF molecules lost in the MT segment gaps.
30	number_of_cargoes	int	An integer variable that stores the number of cargoes present in MT segment
31	m_to_m	int	An integer array of three elements that stores the number of BDNF molecules crossing the MT gaps.
32	calcium_concentration	double	A Double variable that stores the concentration of calcium ions at any distance from the transmitter
DIFFUSION TRANSPORT			
1	D	double	A Double variable that stores the diffusion coefficient of calcium in cytosol
2	B	double	A variable that stores the proportionality constant of the Calcium buffering.

3	x	double	A variable that stores the distance from the transmitter region
4	time	double	A variable that stores the time for diffusion
5	Deff	double	A variable that stores the effective diffusion coefficient of the diffusion transport taking into consideration the calcium buffering phenomenon.
6	expo_term	double	A variable that stores the exponential component of the diffusion equation
7	constant_term	double	A variable that stores the constant part of the diffusion equation
8	value	double	A variable that stores the calculated value of the calcium concentration diffused at distance x from the transmitter

1.2.2 Member functions of TRANSPORT part:

The member function of the active transport part takes in two variables “num_calcium” and “num_BDNF”, generates the number of BDNF molecules lost and served in the MT queue which will be used in the receptor part (gates) and for metric calculation.

Function declaration for active transport is:

int active_transport(int,int);

The member function of the passive transport takes in distance “distance” and “initial_conc” and returns the calcium concentration diffused to that distance by the source.

Function declaration for diffusion transport is:

double diffusion_transport(int,int);

1.3 Receiver part*:*

The variables used in the receiver part of our code can be tabulated as follows:

1.3.1 Data Members of RECEIVER part:

Sl. No.	**Variable Name**	**Variable Data type**	**Variable Description**
SUB GRADIENT OPTIMISATION FUNCTION			
1	zip	double	It stores the value of an overestimate of L(lambda)
2	lambda1	double	One of the lambda multiplier
3	lambda2	double	One of the lambda multiplier
4	lambda3	double	One of the lambda multiplier
5	lambda4	double	One of the lambda multiplier
6	lambda5	double	One of the lambda multiplier
7	lambda6	double	One of the lambda multiplier
8	imp	int	An improvement counter
9	k	int	A loop counter for running the loop
10	best_lambda1	double	A variable that stores the best value of lambda1
11	best_lambda2	double	A variable that stores the best value of lambda2
12	best_lambda3	double	A variable that stores the best value of lambda3
13	best_lambda4	double	A variable that stores the best value of lambda4
14	best_lambda5	double	A variable that stores the best value of lambda5
15	best_lambda6	double	A variable that stores the best value of lambda6
16	delta	float	floating point variable that stores the step-size of the optimization problem
17	c1	double	Stores the capacity of one of the gates
18	c2	double	Stores the capacity of one of the gates
19	c3	double	Stores the capacity of one of the gates
20	c4	double	Stores the capacity of one of the gates
21	c5	double	Stores the capacity of one of the gates
22	c6	double	Stores the capacity of one of the gates
23	r1	double	Stores the clearance rate of one of the gates
24	r2	double	Stores the clearance rate of one of the gates
25	r3	double	Stores the clearance rate of one of the gates

26	r4	double	Stores the clearance rate of one of the gates
27	r5	double	Stores the clearance rate of one of the gates
28	r6	double	Stores the clearance rate of one of the gates
29	a	int	An integer variable that stores the network complexity parameter
30	tk	double	It stores the value of error tolerance in the optimization problem
31	L_lambda	double	A floating point variable for storing the value of Lagrangian relaxed function
32	G1	double	Stores the value of Sub gradient of gate type 1
33	G2	double	Stores the value of Sub gradient of gate type 2
34	G3	double	Stores the value of Sub gradient of gate type 3
35	G4	double	Stores the value of Sub gradient of gate type 4
36	G5	double	Stores the value of Sub gradient of gate type 5
37	G6	double	Stores the value of Sub gradient of gate type 6
38	fk	double	Stores the value of our objective function
39	W	double	Stores the value of number of BDNF molecules reaching the front-end of the receiver
40	x1	double	Stores the value of the number of gates of gate type 1
41	x2	double	Stores the value of the number of gates of gate type 2
42	x3	double	Stores the value of the number of gates of gate type 3
43	x4	double	Stores the value of the number of gates of gate type 4
44	x5	double	Stores the value of the number of gates of gate type 5
45	x6	double	Stores the value of the number of gates of gate type 6
46	error	double	Stores the value of precision of calculation generated in each loop
47	n	int	Stores the value of number of gate types
MIN FUNCTION			

1	a	double	Stores the value of one of the two variable for which minimum of the two is to be found
2	b	double	Stores the value of one of the two variable for which minimum of the two is to be found

1.3.2 Member functions of RECEIVER part:

The member function of the subgradient part takes in four variables “val[]”,”wt[]”,”W” and “n”, generates the number of gates of each gate type which will be further used for metric calculation purpose.

Function declaration for subgradient optimisation is:

int subgrad_receiver(int [], int [], int,int);

The member function of the minimum function takes in two numbers “a” and “b” and returns the minimum of these two numbers.

Function declaration for minimum function is:

double min(double,double);

1.4 Metric Calculation part:

The variables used in the metric calculation part of our code can be tabulated as follows:

1.4.1 *Data Members of METRIC CALCULATION part:*

Sl. No.	**Variable Name**	**Variable Data type**	**Variable Description**
1	dia_MT	double	Floating point variable for storing the diameter of the MT
2	area_of_cross_section_of_MT	double	Floating point variable for storing the value of area of cross section of the MT
3	rate_MT	double	Stores the value of rate of flow of cargoes through MT
4	flux_MT	double	Stores the value of the flux flowing through MT
5	c0	double	Initial concentration of Calcium ions generated
6	r11	double	Stores the value of the distance of a point from the transmitter
7	y11	double	Stores the value of the y-coordinate if the distance of the point
8	Jx	double	stores the value of flux flowing through x-direction
9	Jy	double	stores the value of flux flowing through y-direction
10	delay_diffusion	double	Stores the value of the delay occurring in flow due to diffusion
11	delay_MT_track	double	Stores the value of the delay occurring on the MT track

1.4.2 *Member functions of the METRIC CALCULTION part:*

The member function of the metric calculation part takes in no arguments and prints the value of all the metrics of the neuron model for this run of simulation.

Function definition of the metric calculation part is:

int metric_calculation();

2. CODEs for each part of model:

2.1 Transmitter part:

2.1.1 POISSON GENERATION:

```
int poisson_generation(double lamda)
{
double L,p=1;
int k=0;
double u;
L=1/exp(lamda);
do
{
        k=k+1;
        u=rand()%10000;
        u=u/10000;
        p=p*u;
        } while(p>L);
return(k-1);
 }
```

2.1.2 TRANSMITTER PART:

```
int transmitter(int arrival_rate)
{
int number_of_elements_generated = poisson_generation(arrival_rate)*pow(10,-
6)*sqrt(pow(x_distance,2)+pow(y_distance,2))/(2*sqrt(Diffusion_coefficient*3.14)*
pow(time_interval,1.5));;
return number_of_elements_generated;
}
```

2.2 Transport part:

2.2.1 ACTIVE PART:

```
int active_transport(int num_calcium, int num_BDNF)
{
// source to first level of tubules ( fick's law)
        src_n =0 ;
    for(src_n=0;src_n<3;src_n++)
    {
          tot_arv_calcium=tot_arv_calcium+num_calcium;
          tot_arv_BDNF = tot_arv_BDNF+num_BDNF;
     for(arr_pkt=1;arr_pkt<=num_BDNF;arr_pkt++)
       {
         count=0;
         temp= start[src_n];     // checking ip q status
         while(temp!=NULL)
          {
           count++;
           temp=temp->next;
          }

          if(count<b[src_n])    // if buffer is not full
          {
                value=1;
            start[src_n]= makeq(start[src_n],value); // entry in ip q
           }
            if(count>=b[src_n])          // if buffer is full
              tot_lost++;  // packet drop
        }
          }

// cargoes taking ions on microtubule and storing it on other side queues
          double calcium_concentration=0;
     for(Level=0;Level<4;Level++)
         {
          des=6*Level+2;
         for(src=6*Level;src<6*Level+3;src++)
          {
           des++;

calcium_concentration = diffusion_transport(src*10,tot_arv_calcium);
  mean_sw_rate= (v_cargo/l_mt)-20*calcium_concentration;
  number_of_cargoes=
mean_sw_rate*0.9+(mean_sw_rate*0.1)*(rand()%1000)/1000;
      for(sw_car=1;sw_car<=number_of_cargoes;sw_car++)
      {
        for(sw_pkt=1;sw_pkt<=70;sw_pkt++)
```

```
{
    count=0;
    temp= start[src];

   if(start[src]==NULL)
     break;
   if(start[src]!=NULL)               // ip q not empty
   {

          start[src] = deleteq(start[src]);
          temp= start[des];           // checking op q status

    while(temp!=NULL)
    {

        count++;
        temp=temp->next;
    }

    if(count<b[des])     // if buffer not full
      {
          value=1;
        start[des]= makeq(start[des],value);  // entry in op q
      }
    if(count>=b[des])           // if buffer full
        tot_lost++;   // packet drop

                      }
               }
  }
}

if(Level==3)
  continue;

// tubule to tubule (to be added proper mathematical
expression, now just using some probability)
pr[0]=0.5,pr[1]=0.3,pr[2]=0.2;

for(src=6*Level+3;src<6*Level+6;src++)
 {
  count = 0;
  temp=start[src];
  while(temp!=NULL)
  {
     count++;
```

```
temp=temp->next;
             }

for(des=6*Level+6;des<6*Level+9;des++)
{
            gap_index=(des-src)%3;
            m_to_m[gap_index]= (int)(count * pr[gap_index]);

for(sw_pkt=1;sw_pkt<=m_to_m[gap_index];sw_pkt++)
     {
                // remove packet from that
                start[src] = deleteq(start[src]);

                        count2=0;
                        temp= start[des];

                while(temp!=NULL)
          {
           count2++;
           temp=temp->next;
          }

     if(count2<b[des])     // if buffer not full
      {
            value=1;
         start[des]= makeq(start[des],value);  // entry in op q
      }
    if(count2>=b[des])           // if buffer full
         tot_lost++;   // packet drop

                          }

                       }

    }

for(i=des-2;i<=des;i++)
              {
                 while(start[i]!=NULL)
                 {
                    start[i] = deleteq(start[i]);
                                    tot_dep++;
                                  }

                      }
```

```
    }



            for(i=0;i<=des-3;i++)
        {
          temp=start[i];
          while(temp!=NULL)
           {
                    tot_wait++;
                    temp=temp->next;
                                }
                     }
}
```

2.2.2 PASSIVE TRANSPORT:

```
double diffusion_transport(int distance,int initial_conc)
    {
        double D = 1 * pow(10,-9);
        double B = 0.5;
        double x = distance*pow(10,-9);
        double time =  1000* pow(10,-9);
        double Deff = D/(1+B);
        double expo_term = -x*x/(4*Deff*time);
        double constant_term = 1/(pow((4*3.142*Deff*time),1.5));
        double value = initial_conc *
constant_term*exp(expo_term)*(pow(10,-27));
        return value;
    }
```

<u>OTHER METHODS USED IN TRANSPORT PART:</u>

```
//Making a queue
node* makeq (node* start1,int value)
    {
      node* t=(node*)malloc(sizeof(node));
      node* t1;
     t->val= value;

       if(start1==NULL)
         {
          start1=t;
          t1=t;
          t1->next=NULL;
          return (start1);
         }
       else
         {
           t1=start1;

             while(t1->next!=NULL)
              {
                t1=t1->next;
              }

          t1->next=t;

          t->next=NULL;
          return(start1);

         }
```

```
}
// deleting a queue:

node* deleteq (node* start1)
{
  node* t,*t1, *last;
  int a,b;


    t1=start1;
        while(t1->next!=NULL)
            t1=t1->next;

    if(start1==t1)
     {
      t=start1;
      a=t->val;
      x=a;
      free(t);
      start1=NULL;
      return(start1);
     }

    else
     {
      t=start1;
      a=t->val;

      x=a;
      start1=t->next;
      free (t);
      return (start1);}}
//Show a queue
    void showq (node* start1)
{
 node* t=start1;
   if(start1==NULL)
    printf("\nqueue is empty");
   else
  {
   while(t!=NULL)
    {
      printf("%d\n",t->val);

      t=t->next;
    }
```

```
        }
    }
```

2.3 Receiver part:

```
int subgrad_receiver(int val[], int wt[], int W,int n)
{
    double zip=pow(10,20);
    double lambda1 = 20;
    double lambda2 = 120;
    double lambda3 = 14;
    double lambda4 = 33;
    double lambda5 = 1122;
    double lambda6 = 125;
    int k=0,imp=0;
    double best_lambda1=lambda1;
    double best_lambda2=lambda2;
    double best_lambda3=lambda3;
    double best_lambda4=lambda4;
    double best_lambda5=lambda5;
    double best_lambda6=lambda6;
    double best_L_lambda=-10000000;
    float delta = 0.7;
    double x1,x2,x3,x4,x5,x6;
    double c1,c2,c3,c4,c5,c6;
    int a;
    double r1= val[0],r2=val[1],r3=val[2],r4=val[3],r5=val[4],r6=val[5];
    c1=wt[0],c2=wt[1],c3=wt[2],c4=wt[3],c5=wt[4],c6=wt[5];
    double L_lambda=0,fk,tk,G1,G2,G3,G4,G5,G6;
    tk=10;
    a=2;

    do
  {
    abcd :

    imp++;
    k++;



    x1=pow(((r1*lambda1*(c1+c2+c3+c4+c5+c6)/(W*c1)))*W/(r1*a),1/(a-
1))*W/r1;
    x2=pow(((r2*lambda2*(c1+c2+c3+c4+c5+c6)/(W*c2)))*W/(r2*a),1/(a-
1))*W/r2;
    x3=pow(((r3*lambda3*(c1+c2+c3+c4+c5+c6)/(W*c3)))*W/(r3*a),1/(a-
1))*W/r3;
    x4=pow(((r4*lambda4*(c1+c2+c3+c4+c5+c6)/(W*c4)))*W/(r4*a),1/(a-
1))*W/r4;
    x5=pow(((r5*lambda5*(c1+c2+c3+c4+c5+c6)/(W*c5)))*W/(r5*a),1/(a-
1))*W/r5;
```

```
x6=pow(((r6*lambda6*(c1+c2+c3+c4+c5+c6)/(W*c6)))*W/(r6*a),1/(a-1))*W/r6;

L_lambda=pow((r1*x1)/W,a)+pow((r2*x2)/W,a)+pow((r3*x3)/W,a)+pow((r4*x4)/W,a)+pow((r5*x5)/W,a)+pow((r6*x6)/W,a)+lambda1*(1-(r1*x1*(c1+c2+c3+c4+c5+c6)/(W*c1)))+lambda2*(1-(r2*x2*(c1+c2+c3+c4+c5+c6)/(W*c2)))+lambda3*(1-(r3*x3*(c1+c2+c3+c4+c5+c6)/(W*c3)))+lambda4*(1-(r4*x4*(c1+c2+c3+c4+c5+c6)/(W*c4)))+lambda5*(1-(r5*x5*(c1+c2+c3+c4+c5+c6)/(W*c5)))+lambda6*(1-(r6*x6*(c1+c2+c3+c4+c5+c6)/(W*c6)));//+lambda3*((-c1*x1/W-c2*x2/W)+1);
fk=pow((r1*x1)/W,a)+pow((r2*x2)/W,a)+pow((r3*x3)/W,a)+pow((r4*x4)/W,a)+pow((r5*x5)/W,a)+pow((r6*x6)/W,a);

if(L_lambda>best_L_lambda)
{
        best_L_lambda=L_lambda;
        best_lambda1=lambda1;
        best_lambda2=lambda2;
        best_lambda3=lambda3;
        best_lambda4=lambda4;
        best_lambda5=lambda5;
        best_lambda6=lambda6;
        imp=-1;
}


if(x1>0 && x2>0&& x3>0&& x4>0 && x5>0 &&x6>0)
{
        if(fk<zip)
        zip=fk;
}


if(imp>20)
{
        delta=delta/2;
        lambda1=best_lambda1;
        lambda2=best_lambda2;
        lambda3=best_lambda3;
        lambda4=best_lambda4;
        lambda5=best_lambda5;
        lambda6=best_lambda6;
        imp=0;
        goto abcd;
}
```

```
    double error= abs((zip-best_L_lambda)/best_L_lambda);
    if(k>1000000 || delta < 0.0000001 || tk< 0.00000001 || error < 0.00000001)
      break;

    G6 = 1-(r6*x6*(c1+c2+c3+c4+c5+c6)/(W*c6));
    G5 = 1-(r5*x5*(c1+c2+c3+c4+c5+c6)/(W*c5));
    G4 = 1-(r4*x4*(c1+c2+c3+c4+c5+c6)/(W*c4));
    G3 = 1-(r3*x3*(c1+c2+c3+c4+c5+c6)/(W*c3));
    G2 = 1-(r2*x2*(c1+c2+c3+c4+c5+c6)/(W*c2));
    G1 = 1-(r1*x1*(c1+c2+c3+c4+c5+c6)/(W*c1));
    tk= delta * (zip-L_lambda)/(G1*G1+G2*G2+G3*G3+G4*G4+G5*G5+G6*G6);


    lambda1=min(-1,lambda1+(tk*G1));
   lambda2=min(-1,lambda2+(tk*G2));
   lambda3=min(-1,lambda3+(tk*G3));
   lambda4=min(-1,lambda4+(tk*G4));
   lambda5=min(-1,lambda5+(tk*G5));
    lambda6=min(-1,lambda6+(tk*G6));
    }while(true);
printf("\n No. of gates of capacity %f required : %f \n",c1,ceil(x1));
printf("\n No. of gates of capacity %f required : %f \n",c2,ceil(x2));
printf("\n No. of gates of capacity %f required : %f \n",c3,ceil(x3));
printf("\n No. of gates of capacity %f required : %f \n",c4,ceil(x4));
printf("\n No. of gates of capacity %f required : %f \n",c5,ceil(x5));
printf("\n No. of gates of capacity %f required : %f \n",c6,ceil(x6));

loss_subgrad = W-
(c1*ceil(x1)+c2*ceil(x2)+c3*ceil(x3)+c4*ceil(x4)+c5*ceil(x5)+c6*ceil(x6));

fk=pow((r1*x1)/W,a)+pow((r2*x2)/W,a)+pow((r3*x3)/W,a)+pow((r4*x4)/W,a)+po
w((r5*x5)/W,a)+pow((r6*x6)/W,a);

printf("%g\n",fk);
return 0;
   }
double min(double a,double b)
      {
            if(a<b)
            return b;
            else
            return a;
            }
```

2.4 Metrics calculation:

```
int metric_calculation()
{
double dia_MT = 4*pow(10,-9);
double area_of_cross_section_of_MT = 3.142*pow(dia_MT,2)/4;
double rate_MT = 1*pow(10,-6)*70*pow(10,-6)*(pow(10,-9))/v_cargo;
double flux_MT = rate_MT/area_of_cross_section_of_MT;

double c0 =poisson_generation(mean_arr_rate)*pow(10,-6);
double r11= sqrt((x_distance*x_distance)+(y_distance*y_distance));
double Jx = c0*r11*exp(-
r11*r11/(4*Diffusion_coefficient*time_interval))/(4*time_interval*sqrt(Diffusion_co
efficient*time_interval*3.142));
double Jy = c0*r11*exp(-
y11*y11/(4*Diffusion_coefficient*time_interval))/(4*time_interval*sqrt(Diffusion_c
oefficient*time_interval*3.142));
double delay_diffusion = (x_distance*x_distance +
y_distance*y_distance)/(2.0*Diffusion_coefficient);
double delay_MT_track = l_mt/v_cargo * 4;


printf("\ntotal BDNF molecules arrived: %d",tot_arv_BDNF);
printf("\ntotal BDNF molecules served: %d",tot_dep);
printf("\ntotal BDNF molecules waiting in inner stage queues
now: %d",tot_wait);
printf("\n\noptimised number of gates for the served BDNF
molecules.\n");


printf("\ntotal BDNF molecules lost in MT: %d\n",tot_lost);
printf("\ntotal BDNF molecules lost in optimisation part:
%d\n",loss_subgrad);
printf("\ntotal BDNF molecules lost in whole model:
%d\n",tot_lost+loss_subgrad);
printf("\nblocking probability= %lf percent
\n",100*((tot_lost+loss_subgrad)*1.0/tot_arv_BDNF));
printf("\nMessage Deliverability = %lf percent \n",100-
(100*((tot_lost+loss_subgrad)*1.0/tot_arv_BDNF)));
printf("\nfunction = %g \n", fk );
printf("\naverage delay in clearance = %g \n",1/(fk*W));
printf("\nDelay due to diffusion = %g\n",delay_diffusion);
printf("\nDelay due on MT tracks = %g\n",delay_MT_track);
printf("\nDiffusive flux in x direction = %g \n",Jx);
printf("\nDiffusive flux in y direction = %g \n",Jy);
printf("\nDiffusive flux = %g \n",Jx+Jy);
```

```
printf("\nFlux on MT = %g  area = %g  rate =
%g\n",flux_MT,area_of_cross_section_of_MT,rate_MT);
  return 0;
}
```

2.5 Main *function* part:

```
int main()
{
  time_t t;
  srand((unsigned)time(&t));

  for(int loop=0;loop<25;loop++)
  {
   start[loop]=NULL;
   b[loop]=7000;
  }
  for(int T=0;T<15;T++)
  {
 int number_of_calcium_generated= transmitter(mean_arr_rate);
 int number_of_BDNF_generated = transmitter(mean_arr_rate);
 int trans = transport(number_of_calcium_generated,
number_of_BDNF_generated);
  }
  W= tot_dep;
printf("\nTime taken ::: %d unit",subgrad_receiver(val,wt,W,n));
        int metrics =metric_calculation();
  return 0;
}
```

3. WHOLE CODE of our MODEL:

The whole code of our model is presented below:

```
# include<stdio.h>
        # include<time.h>
        # include<stdlib.h>
        # include<math.h>

        typedef struct nodedata{
                        int val;
                struct nodedata *next;
                }node;



        node* makeq (node*,int);
        node* deleteq (node*);
        void showq (node*);
        int poisson_generation(double);
        double diffusion_transport(int,int);
        int active_transport(int,int);
        int subgrad_receiver(int [], int [], int,int);
        double min(double,double);
        int transmitter(int);
        int metric_calculation();

        node* start[25];
        node *temp;
        int
gap_index,i,arr_pkt,sw_car,sw_pkt,src,des,src_n,b[25],count,count2;
        int
tot_arv_calcium=0,tot_arv_BDNF=0,tot_lost=0,tot_dep=0,tot_wait=0,value,Level;
        double mean_arr_rate=20,mean_srv_rate= 60,l_mt=8*pow(10,-
9),v_cargo=800*pow(10,-9),mean_sw_rate;
        double mean_arr_rate_ca=40,mean_srv_rate_ca= 80;
        double pr[3];
        int wt[]={31,38,25,30,48,26};           // Quantity of each gate
    int val[]={25,21,18,29,35,23};        // time taken
    int W;                                          // total packet
served
    int n=6;                                       //No. of cargo
        int number_of_cargoes,m_to_m[3];
        int x;
        int loss_subgrad;
        double fk;
        double x_distance=pow(10,-
9),y_distance=0,Diffusion_coefficient=5.3*pow(10,-10),time_interval=pow(10,-9);
```

```
int active_transport(int num_calcium, int num_BDNF)
{
// source to first level of tubules ( fick's law)
        src_n =0 ;
    for(src_n=0;src_n<3;src_n++)
    {
          tot_arv_calcium=tot_arv_calcium+num_calcium;
          tot_arv_BDNF = tot_arv_BDNF+num_BDNF;
     for(arr_pkt=1;arr_pkt<=num_BDNF;arr_pkt++)
       {
         count=0;
         temp= start[src_n];     // checking ip q status
         while(temp!=NULL)
          {
           count++;
           temp=temp->next;
          }

          if(count<b[src_n])    // if buffer is not full
          {
                value=1;
            start[src_n]= makeq(start[src_n],value); // entry in ip q
           }
            if(count>=b[src_n])          // if buffer is full
             tot_lost++;  // packet drop
        }
          }

          // cargoes taking ions on microtubule and storing it on
other side queues
          double calcium_concentration=0;
   for(Level=0;Level<4;Level++)
         {
          des=6*Level+2;
         for(src=6*Level;src<6*Level+3;src++)
          {
           des++;

           calcium_concentration =
diffusion_transport(src*10,tot_arv_calcium);
           mean_sw_rate= (v_cargo/l_mt)-
20*calcium_concentration;
```

```
                                        number_of_cargoes=
mean_sw_rate*0.9+(mean_sw_rate*0.1)*(rand()%1000)/1000;
                            for(sw_car=1;sw_car<=number_of_cargoes;sw_car++)
                            {
                              for(sw_pkt=1;sw_pkt<=70;sw_pkt++)
                              {
                                 count=0;
                                 temp= start[src];

                                if(start[src]==NULL)
                                 break;
                                if(start[src]!=NULL)                // ip q not empty
                                {

                                       start[src] = deleteq(start[src]);
                                       temp= start[des];            // checking op q status

                                 while(temp!=NULL)
                                 {

                                    count++;
                                    temp=temp->next;
                                 }

                                 if(count<b[des])     // if buffer not full
                                   {
                                       value=1;
                                     start[des]= makeq(start[des],value);  // entry in op q
                                   }
                                 if(count>=b[des])           // if buffer full
                                     tot_lost++;   // packet drop

                                                   }
                                             }
                             }
                          }

                          if(Level==3)
                            continue;

                          // tubule to tubule (to be added proper mathematical
expression, now just using some probability)
                          pr[0]=0.5,pr[1]=0.3,pr[2]=0.2;

                         for(src=6*Level+3;src<6*Level+6;src++)
                          {
```

```
                    count = 0;
                    temp=start[src];
                    while(temp!=NULL)
                    {
                      count++;
                      temp=temp->next;
                               }

                      for(des=6*Level+6;des<6*Level+9;des++)
                      {
                              gap_index=(des-src)%3;
                              m_to_m[gap_index]= (int)(count * pr[gap_index]);

for(sw_pkt=1;sw_pkt<=m_to_m[gap_index];sw_pkt++)
                         {
                                 // remove packet from that
                                 start[src] = deleteq(start[src]);

                                       count2=0;
                                       temp= start[des];

                                  while(temp!=NULL)
                              {
                               count2++;
                               temp=temp->next;
                              }

                          if(count2<b[des])     // if buffer not full
                           {
                               value=1;
                             start[des]= makeq(start[des],value);  // entry in op q
                           }
                         if(count2>=b[des])           // if buffer full
                             tot_lost++;   // packet drop

                                          }

                                        }

                        }

              for(i=des-2;i<=des;i++)
                    {
                      while(start[i]!=NULL)
                      {
                        start[i] = deleteq(start[i]);
```

```
                        tot_dep++;
                       }

                  }

         }



             for(i=0;i<=des-3;i++)
           {
            temp=start[i];
            while(temp!=NULL)
             {
                 tot_wait++;
                 temp=temp->next;
                       }
                  }

       }

       double diffusion_transport(int distance,int initial_conc)
       {
            double D = 1 * pow(10,-9);
            double B = 0.5;
            double x = distance*pow(10,-9);
            double time =  1000* pow(10,-9);
            double Deff = D/(1+B);
            double expo_term = -x*x/(4*Deff*time);
            double constant_term = 1/(pow((4*3.142*Deff*time),1.5));
            double value = initial_conc *
constant_term*exp(expo_term)*(pow(10,-27));
            return value;
       }


            int poisson_generation(double lamda)
       {
            double L,p=1;
            int k=0;
            double u;
            L=1/exp(lamda);
            do
            {
                 k=k+1;
                 u=rand()%10000;
```

```
u=u/10000;
p=p*u;
} while(p>L);

return(k-1);

}


int transmitter(int arrival_rate)
{
int number_of_elements_generated =
poisson_generation(arrival_rate)*pow(10,-
6)*sqrt(pow(x_distance,2)+pow(y_distance,2))/(2*sqrt(Diffusion_coefficient*3.14)*
pow(time_interval,1.5));;
return number_of_elements_generated;
}

int main()
{
  time_t t;
  srand((unsigned)time(&t));

  for(int loop=0;loop<25;loop++)
  {
   start[loop]=NULL;
   b[loop]=7000;
  }
  for(int T=0;T<15;T++)
  {
          int number_of_calcium_generated=
transmitter(mean_arr_rate_ca);
            int number_of_BDNF_generated =
transmitter(mean_arr_rate);
            int trans = transport(number_of_calcium_generated,
number_of_BDNF_generated);
  }
            W= tot_dep;
            printf("\nTime taken ::: %d
unit",subgrad_receiver(val,wt,W,n));
            int metrics =metric_calculation();
  return 0;

}

int metric_calculation()
```

```
{
        double dia_MT = 4*pow(10,-9);
       double area_of_cross_section_of_MT =
3.142*pow(dia_MT,2)/4;
       double rate_MT = 1*pow(10,-6)*70*pow(10,-6)*(pow(10,-
9))/v_cargo;
       double flux_MT = rate_MT/area_of_cross_section_of_MT;

    double c0 =poisson_generation(mean_arr_rate)*pow(10,-6);
        double r11=
sqrt((x_distance*x_distance)+(y_distance*y_distance));
        double Jx = c0*r11*exp(-
r11*r11/(4*Diffusion_coefficient*time_interval))/(4*time_interval*sqrt(Diffusion_co
efficient*time_interval*3.142));
        double Jy = c0*r11*exp(-
y11*y11/(4*Diffusion_coefficient*time_interval))/(4*time_interval*sqrt(Diffusion_c
oefficient*time_interval*3.142));
        double delay_diffusion = (x_distance*x_distance +
y_distance*y_distance)/(2.0*Diffusion_coefficient);
        double delay_MT_track = l_mt/v_cargo * 4;


      printf("\ntotal BDNF molecules arrived: %d",tot_arv_BDNF);
      printf("\ntotal BDNF molecules served: %d",tot_dep);
      printf("\ntotal BDNF molecules waiting in inner stage queues
now: %d",tot_wait);
      printf("\n\noptimised number of gates for the served BDNF
molecules.\n");


      printf("\ntotal BDNF molecules lost in MT: %d\n",tot_lost);
      printf("\ntotal BDNF molecules lost in optimisation part:
%d\n",loss_subgrad);
      printf("\ntotal BDNF molecules lost in whole model:
%d\n",tot_lost+loss_subgrad);
      printf("\nblocking probability= %lf percent
\n",100*((tot_lost+loss_subgrad)*1.0/tot_arv_BDNF));
      printf("\nMessage Deliverability = %lf percent \n",100-
(100*((tot_lost+loss_subgrad)*1.0/tot_arv_BDNF)));
      printf("\nfunction = %g \n", fk );
  printf("\naverage delay in clearance = %g \n",1/(fk*W));
  printf("\nDelay due to diffusion = %g\n",delay_diffusion);
  printf("\nDelay due on MT tracks = %g\n",delay_MT_track);
  printf("\nDiffusive flux in x direction = %g \n",Jx);
      printf("\nDiffusive flux in y direction = %g \n",Jy);
      printf("\nDiffusive flux = %g \n",Jx+Jy);
```

```
printf("\nFlux on MT = %g  area = %g  rate =
%g\n",flux_MT,area_of_cross_section_of_MT,rate_MT);
    return 0;
  }

         node* makeq (node* start1,int value)
  {
     node* t=(node*)malloc(sizeof(node));
     node* t1;
    t->val= value;

      if(start1==NULL)
        {
         start1=t;
         t1=t;
         t1->next=NULL;
         return (start1);
        }
      else
        {
          t1=start1;

            while(t1->next!=NULL)
             {
               t1=t1->next;
             }

         t1->next=t;

         t->next=NULL;
         return(start1);

        }


  }

  node* deleteq (node* start1)
  {
    node* t,*t1, *last;
    int a,b;


     t1=start1;
         while(t1->next!=NULL)
              t1=t1->next;
```

```
     if(start1==t1)
      {
       t=start1;
       a=t->val;
       x=a;
       free(t);
       start1=NULL;

       //printf("\ndeleted value=%d",a);
       return(start1);
      }

     else
      {
       t=start1;
       a=t->val;

       x=a;
       start1=t->next;
       free(t);
      // printf("\ndeleted value=%d",b);
       return(start1);

      }

}

void showq (node* start1)
{
  node* t=start1;
    if(start1==NULL)
     printf("\nqueue is empty");
    else
   {
    while(t!=NULL)
     {
        printf("%d\n",t->val);

        t=t->next;
     }
    }
}
```

```
 int subgrad_receiver(int val[], int wt[], int W,int n)
{
      double zip=pow(10,20);
      double lambda1 = 20;
      double lambda2 = 120;
      double lambda3 = 14;
      double lambda4 = 33;
      double lambda5 = 1122;
      double lambda6 = 125;
      int k=0,imp=0;
      double best_lambda1=lambda1;
      double best_lambda2=lambda2;
      double best_lambda3=lambda3;
    double best_lambda4=lambda4;
    double best_lambda5=lambda5;
      double best_lambda6=lambda6;
      double best_L_lambda=-10000000;
      float delta = 0.7;
      double x1,x2,x3,x4,x5,x6;
      double c1,c2,c3,c4,c5,c6;
      int a;
      double r1= val[0],r2=val[1],r3=val[2],r4=val[3],r5=val[4],r6=val[5];
      c1=wt[0],c2=wt[1],c3=wt[2],c4=wt[3],c5=wt[4],c6=wt[5];
      double L_lambda=0,fk,tk,G1,G2,G3,G4,G5,G6;
      tk=10;
      a=2;

      do
   {
      abcd :

      imp++;
      k++;


      x1=pow((((r1*lambda1*(c1+c2+c3+c4+c5+c6)/(W*c1)))*W/(r1*a),1/(a-
1))*W/r1;
      x2=pow((((r2*lambda2*(c1+c2+c3+c4+c5+c6)/(W*c2)))*W/(r2*a),1/(a-
1))*W/r2;
      x3=pow((((r3*lambda3*(c1+c2+c3+c4+c5+c6)/(W*c3)))*W/(r3*a),1/(a-
1))*W/r3;
      x4=pow((((r4*lambda4*(c1+c2+c3+c4+c5+c6)/(W*c4)))*W/(r4*a),1/(a-
1))*W/r4;
      x5=pow((((r5*lambda5*(c1+c2+c3+c4+c5+c6)/(W*c5)))*W/(r5*a),1/(a-
1))*W/r5;
```

```
    x6=pow((((r6*lambda6*(c1+c2+c3+c4+c5+c6)/(W*c6)))*W/(r6*a),1/(a-1))*W/r6;

    L_lambda=pow((r1*x1)/W,a)+pow((r2*x2)/W,a)+pow((r3*x3)/W,a)+pow((r4*x4)/W,a)+pow((r5*x5)/W,a)+pow((r6*x6)/W,a)+lambda1*(1-(r1*x1*(c1+c2+c3+c4+c5+c6)/(W*c1)))+lambda2*(1-(r2*x2*(c1+c2+c3+c4+c5+c6)/(W*c2)))+lambda3*(1-(r3*x3*(c1+c2+c3+c4+c5+c6)/(W*c3)))+lambda4*(1-(r4*x4*(c1+c2+c3+c4+c5+c6)/(W*c4)))+lambda5*(1-(r5*x5*(c1+c2+c3+c4+c5+c6)/(W*c5)))+lambda6*(1-(r6*x6*(c1+c2+c3+c4+c5+c6)/(W*c6)));//+lambda3*((-c1*x1/W-c2*x2/W)+1);
    fk=pow((r1*x1)/W,a)+pow((r2*x2)/W,a)+pow((r3*x3)/W,a)+pow((r4*x4)/W,a)+pow((r5*x5)/W,a)+pow((r6*x6)/W,a);

    if(L_lambda>best_L_lambda)
    {
            best_L_lambda=L_lambda;
            best_lambda1=lambda1;
            best_lambda2=lambda2;
            best_lambda3=lambda3;
            best_lambda4=lambda4;
            best_lambda5=lambda5;
            best_lambda6=lambda6;
            imp=-1;
    }


    if(x1>0 && x2>0&& x3>0&& x4>0 && x5>0 &&x6>0)
    {
            if(fk<zip)
            zip=fk;
   }


    if(imp>20)
    {
            delta=delta/2;
            lambda1=best_lambda1;
            lambda2=best_lambda2;
            lambda3=best_lambda3;
            lambda4=best_lambda4;
            lambda5=best_lambda5;
            lambda6=best_lambda6;
            imp=0;
            goto abcd;
    }
```

```
      double error= abs((zip-best_L_lambda)/best_L_lambda);
      if(k>1000000 || delta < 0.0000001 || tk< 0.00000001 || error < 0.00000001)
        break;

      G6 = 1-(r6*x6*(c1+c2+c3+c4+c5+c6)/(W*c6));
      G5 = 1-(r5*x5*(c1+c2+c3+c4+c5+c6)/(W*c5));
      G4 = 1-(r4*x4*(c1+c2+c3+c4+c5+c6)/(W*c4));
      G3 = 1-(r3*x3*(c1+c2+c3+c4+c5+c6)/(W*c3));
      G2 = 1-(r2*x2*(c1+c2+c3+c4+c5+c6)/(W*c2));
      G1 = 1-(r1*x1*(c1+c2+c3+c4+c5+c6)/(W*c1));
      tk= delta * (zip-L_lambda)/(G1*G1+G2*G2+G3*G3+G4*G4+G5*G5+G6*G6);


      lambda1=min(-1,lambda1+(tk*G1));
     lambda2=min(-1,lambda2+(tk*G2));
     lambda3=min(-1,lambda3+(tk*G3));
     lambda4=min(-1,lambda4+(tk*G4));
     lambda5=min(-1,lambda5+(tk*G5));
      lambda6=min(-1,lambda6+(tk*G6));
      }while(true);


printf("\n No. of gates of capacity %f required : %f \n",c1,ceil(x1));
printf("\n No. of gates of capacity %f required : %f \n",c2,ceil(x2));
printf("\n No. of gates of capacity %f required : %f \n",c3,ceil(x3));
printf("\n No. of gates of capacity %f required : %f \n",c4,ceil(x4));
printf("\n No. of gates of capacity %f required : %f \n",c5,ceil(x5));
printf("\n No. of gates of capacity %f required : %f \n",c6,ceil(x6));

loss_subgrad = W-
(c1*ceil(x1)+c2*ceil(x2)+c3*ceil(x3)+c4*ceil(x4)+c5*ceil(x5)+c6*ceil(x6));

fk=pow((r1*x1)/W,a)+pow((r2*x2)/W,a)+pow((r3*x3)/W,a)+pow((r4*x4)/W,a)+po
w((r5*x5)/W,a)+pow((r6*x6)/W,a);

printf("%g\n",fk);

return 0;
    }


         double min(double a,double b)
         {
                if(a<b)
```

```
return b;
else
return a;
}
```