Inhaltsverzeichnis

1 Einleitung **1**

2 Auryn **1**

3 Design **2**

4 Installation **4**

5 Benutzung **4**

6 Korrektheit **6**

7 Feature Requests **6**

1 Einleitung

Angestrebt wurde eine Sofware zur Simulation von Neuronalen Netzen im primären visuellen Cortex. Eine Besonderheit der Netze in diesem Bereich ist die Art ihrer Verbindungen untereinander:

- Die Verbindungen entstehen überwiegend lokal.
- Neuronen mit ähnlicher Orientierungsselektivität können auch über größere Distanzen Verbindungen zueinander aufbauen.

Ein Modell, welches diese Besonderheiten abbildet, wird in Abbildung 1 beschrieben.

Um die Simulationen möglichst performant zu halten wurde auf das in C++ entwickelte Framework Auryn [1] aufgebaut.

2 Auryn

Auryn [1] ist ein Framework zur Simulierung von plastischen neuronalen Netzwerken. Jede Auryn Simulation besteht im wesentlichen aus vier Modulen:

- Die *SpikingGroup* Objekte erzeugen je nach Verwendung zeitlich organisierte Stimuli und bilden so die Eingabe in das gewünschte Netzwerk.
- Die *NeuronGroup* Objekte bilden die Neuronenpopulationen ab und berechnen den aktuellen Potentialwert.
- Die *Connection* Objekte sorgen für die Verbindung der Neuronen untereinander. Sie übernehmen die Propagation der Signale.
- Die *Monitor* Objekte zeichnen zu gewünschten Zeitpunkten gewünschte Signale auf, z.B. zur Evaluation von Netzwerkstati.

3 Design

Auryn stellt bereits die gewünschten Neuronenmodelle bereit. Nur eine Verbindung zwischen den Neuronen nach dem Modell des primären visuellen Cortex ist in der gewünschte Form nicht vorhanden. Um diese zu modellieren wurde eine neue Klasse *GeoConnection* entwickelt, welche direkt auf der von Auryn bereitgestellten *SparseConnection* Klasse aufbaut (erbt). Die Klasse *GeoConnection* bringt drei wichtige Funktionen mit:

1. Die Berechnung der X und Y Position im 2D-Gitter jedes Neurons:

```
void get_xy(NeuronID id, double &x, double &y, bool source){
  id += 1;   // NeuronID starts with 0
  if (source) {
      unsigned int r = id % sourceWidth;
      x = (r==0)? sourceWidth + sourceGap : r + sourceGap;
      y = floor((id-x)/sourceWidth)+1+sourceGap;
  }else {
      unsigned int r = id % destWidth;
      x = (r==0)? destWidth + destGap : r + destGap;
      y = floor((id-x)/destWidth)+1+destGap;
  }
}
```

   Beim Erzeugen von Neuronenpopulationen nummeriert Auryn diese mit einer eindeutigen ID durch, beginnend mit 0. Durch eine Angabe der Gitterbreite (...Width) und einem Startwert (...Gap) wird die jeweilige Position des Neurons mit der ID id berechnet. Durch den Parameter source können zwei verschiedene Gitter für Quell und Zielpopulation benutzt werden. Um sich die Rückgabe über einen Vektor zu sparen müssen x und y vor Aufruf der Funktion angelegt werden um deren Adresse bei Funktionsaufruf mitzugegeben.

2. Die Berechnung der Orientierungsselektivität:

```
unsigned short get_orientation(double x, double y) {
    double b = (double) oBoxSize;
    double m = (double) round(b/2);

    // calculate quadrant
    unsigned int fx = floor((x-1)/b);
    unsigned int fy = floor((y-1)/b);

    // Center Point of quadrant
    double mx = m + fx * b;
    double my = m + fy * b;

    // center has 0 degree
    if(mx==x && my == y) return 0;

    // determine rotation direction
    short clockwise = (fx^fy)? -1 : 1;

    // calculate angle
    double a = (atan2((my-y),(mx-x))*180) / PI -90;
    int o = clockwise * a + 0.5;

    // return value between 0-360 degree
    return (o < 0)? o + 360: o;
}
```

Über die Angabe eines Wertes für die Größe der Pinwheel-Felder (oBoxSize) wird ersteinmal der Quadrant (fx,fy) in dem sich das Neuron befindet bestimmt. Mit diesem Wissen kann sowohl der Mittelpunkt (mx,my) des Quadranten wie auch die Richtung der Selektivität (s) bestimmt werden. Als letzten Schritt wird mit der atan2 Funktion der Winkel zwischen Lot und Neuron bestimmt und in Grad umgerechnet. Der Winkel wird als unsigned short im Bereich 0-360 zurückgegeben.

3. Die Bestimmung der Wahrscheinlichkeit für das Zustandekommen einer Verbindung zwischen zwei Neuronen:

```
double getProbability(NeuronID i, NeuronID j, double sigma) {
  double xi,yi,xj,yj;
  get_xy(i,xi,yi,true);
  get_xy(j,xj,yj,false);

  if (same_orientation(xi,yi,xj,yj))
     return this->oWeight;           // Fix weight

  double x = pow(xj-xi,2);
  double y = pow(yj-yi,2);
  return exp(-1*(x+y) / sigma);
}
```

Nach dem die x und y Koordinaten für beide Neuronen bestimmt wurden wird geprüft, ob beide eine ähnliche Orientierungsselektivität haben. Ist dies der Fall wird eine festgelegte Wahrscheinlichkeit zurückgegeben. Ist dies nicht der Fall wird die Wahrscheinlichkeit über die Distanz zwischen beiden Neuronen bestimmt:

$$p(i,j) = e^{\frac{-((x_i-x_j)^2+(y_i-y_j)^2)}{\sigma}}$$

Das Ergebnis wird als double zurückgegeben.

Erzeugt werden die Verbindungen in der Funktion *connectLikeInVC_Block()*. Um eine verteilte Berechnung der Verbindungen zu ermöglichen integriert Auryn die von Boost umgesetzte Schnittstelle zu openMPI und anderen MPI Bibliotheken. Um die Zuständigkeit innerhalb der Verbindungsmatrix in dem jeweiligen Thread zu bestimmen dienen die Variablen r und s:

```
r = communicator->rank() - dst->get_locked_rank();
s = dst->get_locked_range();
```

Aus diesen und einem Counter x, der über alle Matrixeinträge läuft, werden gültige Positionen in der Matrix bestimmt in welche das jeweilige Gewicht eingtragen wird. Um die oben beschriebene Wahrscheinlichkeitsfunktion zu integrieren wird ein Generator auf einer Gleichverteilung initialisiert. Jeder Ziehung (*die()*) wird mit der berechneten Wahrscheinlichkeit verglichen und dementsprechen die Verbindung etabliert oder nicht:

```
  boost::random::uniform_int_distribution<> dist(0,100);
  boost::variate_generator<boost::mt19937&,boost::random::uniform_int_distribution<> >
        die(GeoConnection::sparse_connection_gen, dist);

 [...]

while ( x < idim * jdim ) {
  i = lo_row + x / jdim;
  j = lo_col + s*(x % jdim) + r;

  [...]

  double p = getProbability(i,j,sigma)*100;
  double ps = die();

  if ( (j >= lo_col) && (!skip_diag || i!=j) && p > ps) {
    temp_weight = (same_orientation(i,j))? oWeight: cWeight;
    if ( push_back(i,j,temp_weight) )
  count++;

    x++;
  }
```

4 Installation

Benötigt werden die Biliotheken *Boost* und eine MPI Biliothek wie *openMPI*. Bei beiden Biliotheken werden die Entwickler-Header Dateien benötigt: *boost-devel und openmpi-devel*.

Boost sollte die Pakete *boost_ program_ options, boost_ mpi, boost_ serialization, boost_ filesystem, boost_ system* enthalten.

Zur Nutzung von openMPI muss noch das entsprechende Modul geladen werden, z.B.:

```
module load mpi/openmpi-x86_64
```

5 Benutzung

Zur Benutzung der entwickelten Verbindung kann am besten eine vorhandene Simulation aus dem Ordner examples/ kopiert und entsprechend angepasst werden. Als Beispiel dient hier die datei „sim_vc_test.cpp“ welche im folgenen kurz erläutert wird:

- Festlegen der Netzwerkparameter:

```
      #define NE 4096 //64   // Nummer e Neuronen
      #define NI 1024 //16   // Nummer i Neuronen (4:1)

      #define FW 64    // Gitterbreite
      #define PB 32    // Pinwheel-Box Breite
```

- Festlegen der Gewichte:

```
1       double wEE = 0.001; // Weight for EE connection
2       double wEI = 0.1;   // Weight for EI connection
3       double wIE = 0.1;   // Weight for IE connection
4       double wII = 0.1;   // Weight for II connection
5       double wLR = 0.02;  // Weight for long-range
```

- MPI starten:

```
1       mpi::environment env(ac, av);
2       mpi::communicator world;
3       communicator = &world;
```

- Neuronenpopulationen erstellen:

```
1       // Create NeuronGroup
2       AdExGroup * neurons_e = new AdExGroup(NE);
3       AdExGroup * neurons_i = new AdExGroup(NI);

5       // initial membrane potentials with a Gaussian:
6       // random_mem(mean,sigma)
7       neurons_e->random_mem(-60e-3,5e-3);
8       neurons_i->random_mem(-60e-3,5e-3);
```

- Verbindungen mit der neuen GeoConnection erstellen:

```
1       GeoConnection * con_ee = new GeoConnection(neurons_e, neurons_e, wEE, wLR, false,false,
              FW,sigma,GLUT, "EEConnection");

3       GeoConnection * con_ei = new GeoConnection(neurons_e, neurons_i, wEI, wLR, false,true,FW
              ,sigma,GLUT, "EIConnection");

5       GeoConnection * con_ii = new GeoConnection(neurons_i, neurons_i, wII, wLR, true,true,FW,
              sigma,GABA, "IIConnection");

7       GeoConnection * con_ie = new GeoConnection(neurons_i, neurons_e, wIE, wLR, true,false,FW
              ,sigma,GABA, "IEConnection");
```

  Wobei die Parameter für die Verbindung (*source, destination, Gewicht, Long-Range-Gewicht, source inhibitorisch?, destination inhibitorisch?, Gittergröße, sigma, Transmitter, Name*) lauten.

- Erstellen von Monitoren zur Aufzeichnung von Netzwerkstati:

```
1       SpikeMonitor * smon_e = new SpikeMonitor( neurons_e , strbuf.c_str() );
2       SpikeMonitor * smon_i = new SpikeMonitor( neurons_i, strbuf.c_str() );
```

Die Simulationsdatei sollte im examples/ Order liegen und das Präfix sim_ haben. Wechselt man in den Ordner build/home/ kann das Projekt mit *make* gebaut werden. Vom Wurzelknoten aus kann auch das Skript „build.sh“ benutzt werden.

Gestartet wird die Simulation dann über

```
./build/home/sim_vc_test
```

oder als verteilte Anwendung mittels

```
mpirun -n 4 ./sim_vc_test --dir /tmp --simtime 50
```

wobei die 4 die Anzahl der Threads, also idealerweise die Anzahl der vorhandenen Kerne spezifiziert.

6 Korrektheit

Da die Erzeugung der Verbindungen auf Wahrscheinlichkeiten basiert wurden nur die einzelnen Methoden *getXY(), get_ orientation(), getProbability() & same_ orientation()* auf Korrektheit gestestet. Das Abschließende Ergebnis wurde mit Matlab grafisch evaluiert (siehe Abbildunge 2-4).

7 Feature Requests

- Der Programmcode wurde auf Effizienz optimiert. Ein noch bestehendes Bottleneck ist die Berechnung der Orientierungsselektivität. Diese wird für die Bestimmung der Wahrscheinlichkeit und zur Bestimmung des Gewichtes gebraucht und im Augenblick zweimalig berechnet. Ausserdem kann durch die verteilte Verarbeitung auf n Prozessen theoretisch ein Worst Case von $n * 2$ Berechnungen pro Neuron entstehen. Das Zwischenspeichern der Werte ist durch die verteilte Berechnung nicht ohne großen Performanceverlust realisierbar. Besser wäre eine Abänderung der Methode *getProbability*, so dass die Orientierung zusammen mit der Wahrscheinlichkeit zurückgegeben wird.
- Die Entwicklung einer SpikingGroup, welche mit openCV Sinuide Funktionen und natürliche Videos als Eingabe zu beliebigen Neuronenpopulationen erlaubt.
- Die Entwicklung eines fMRTMonitors, welcher aus einer Simulation eine Schätzung einer fMRT-Messung generiert.

Literatur

[1] Friedemann Zenke and Wulfram Gerstner. Limits to high-speed simulations of spiking neural networks using general-purpose computers. *Frontiers in Neuroinformatics*, 8(76), 2014.

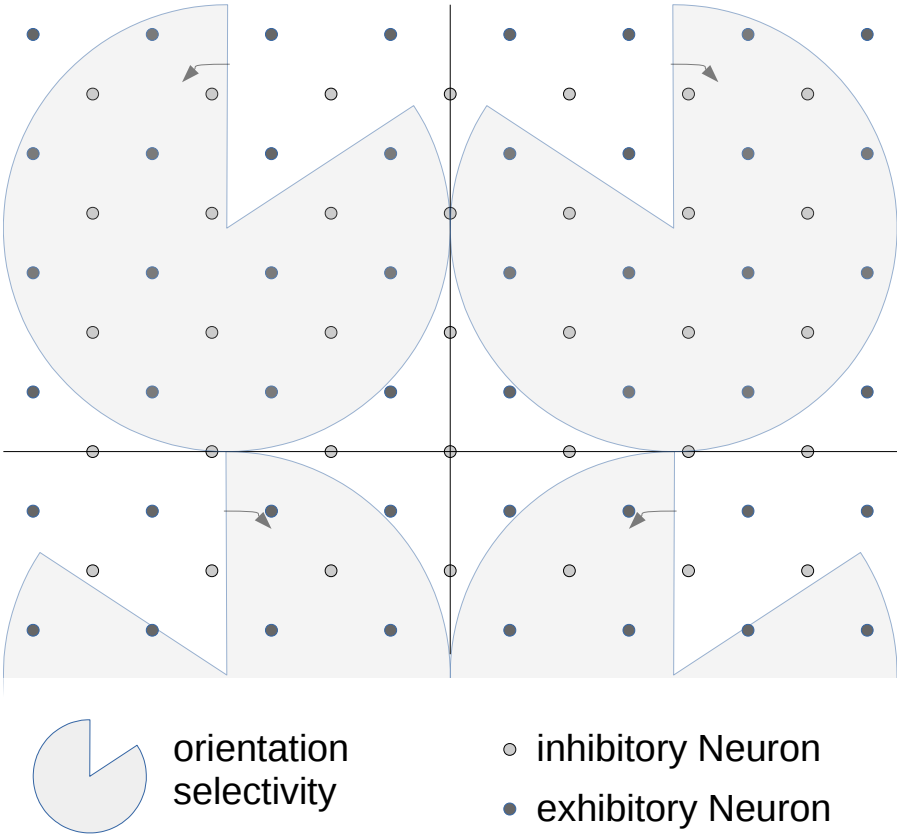
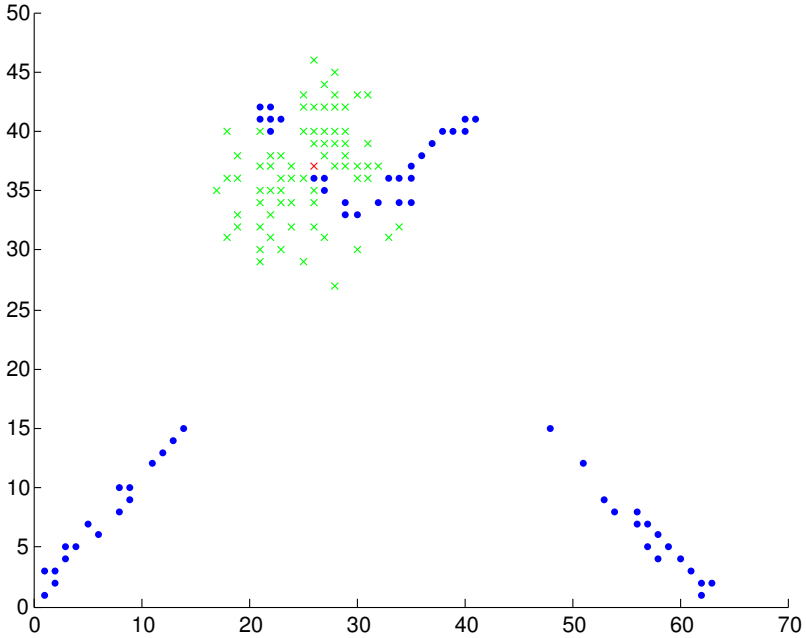
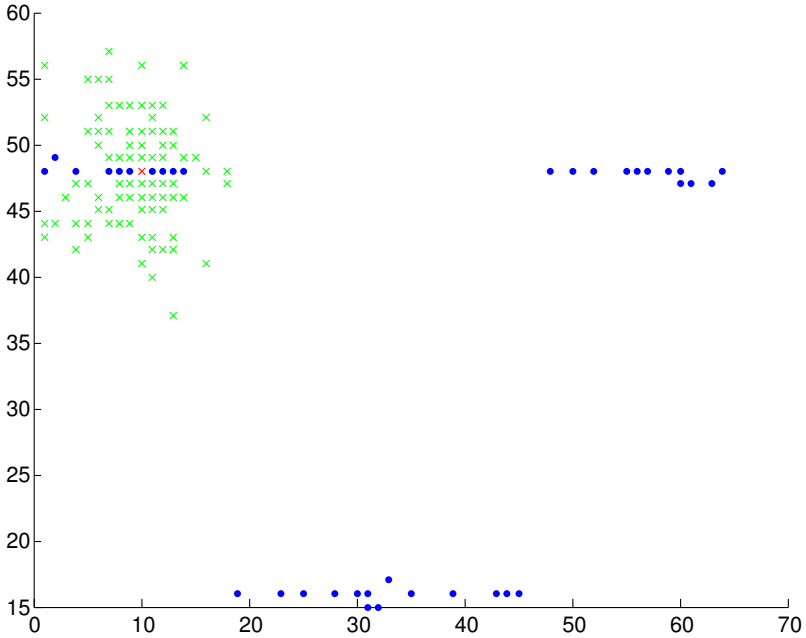


Abbildung 1: Design eines Modells des primären visuellen Cortex. Das Netzwerk besteht zu vier Teilen aus exhibitorischen und zu einem Teil aus inhibitorischen Neuronen. Die Neuronen sind dabei auf einem zweidimensionalem Gitter verteilt. Die inhibitorischen Neuronen liegen jeweils zentral zu vier exihibitorischen Zellen. Neuronale Netzwerke im V1 weißen üblicherweise eine Orientierungsselektivität auf. Wärend die meisten Verbindungen zwischen Neuronen lokal, also über kurze Distanzen hin stattfinden, gibt es zwischen Neuronen mit sehr ähnlicher Selektivität auch über weite Strecken Verbindungen. In diesem Modell wird das Gitter in quadratische Bereiche gegliedert in denen jeweils ein sogenanntes Pinwheel abgebildet wird. Das Lot zum Mittelpunkt bildet jeweils die Null Grad Selektivität. Der Winkel der Orientierung erhöht sich je quadrant abwechselnd im und gegen den Uhrzeigersinn, gelesen von oben links nach unten rechts, wobei im ersten Quadrant gegen den Uhrzeigersinn gestartet wird.

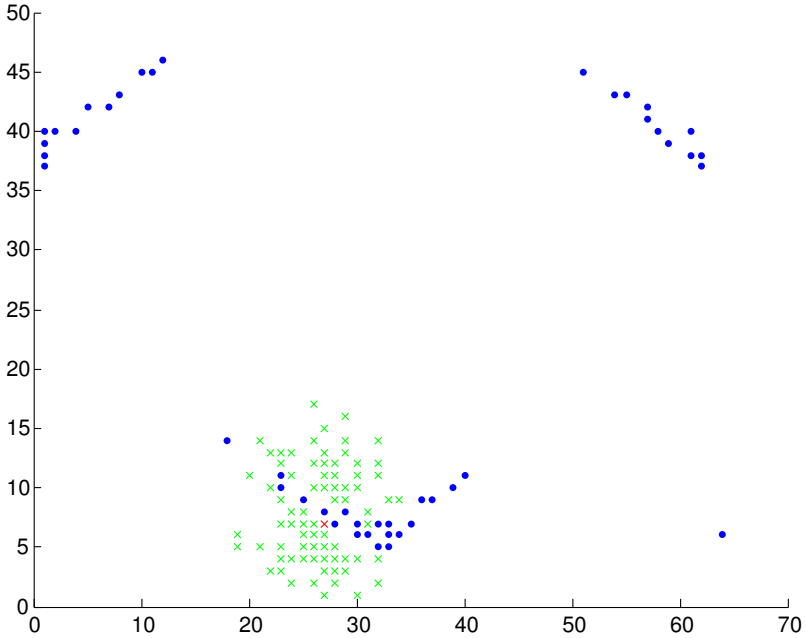


(a)

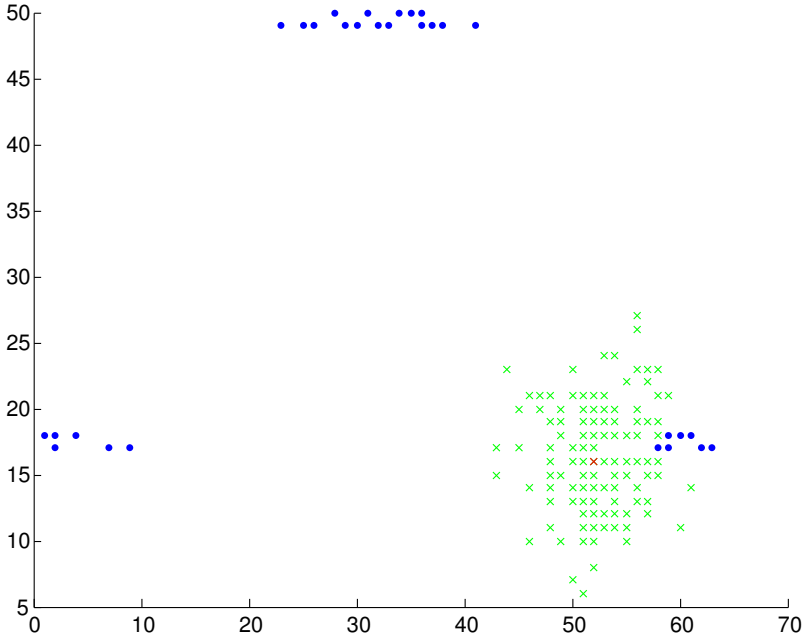


(b)

Abbildung 2: EE Verbindungen

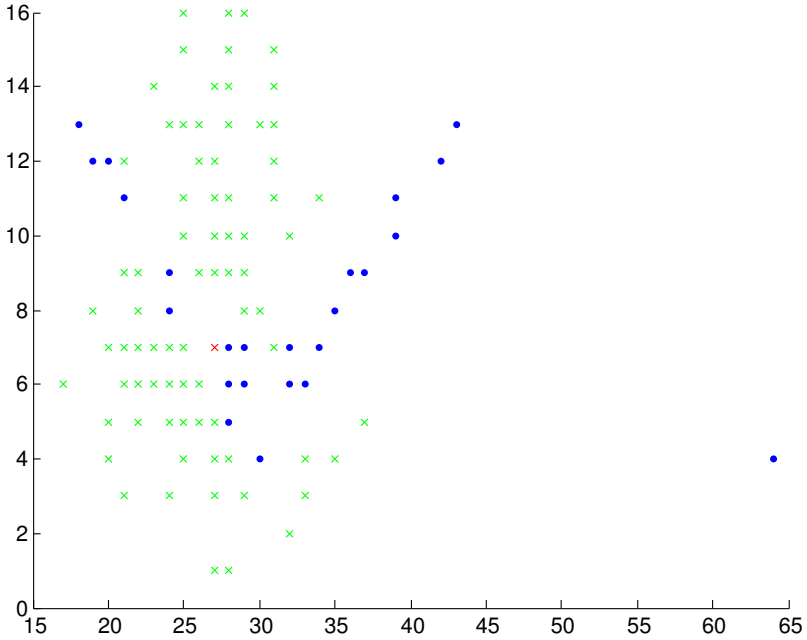


(a)

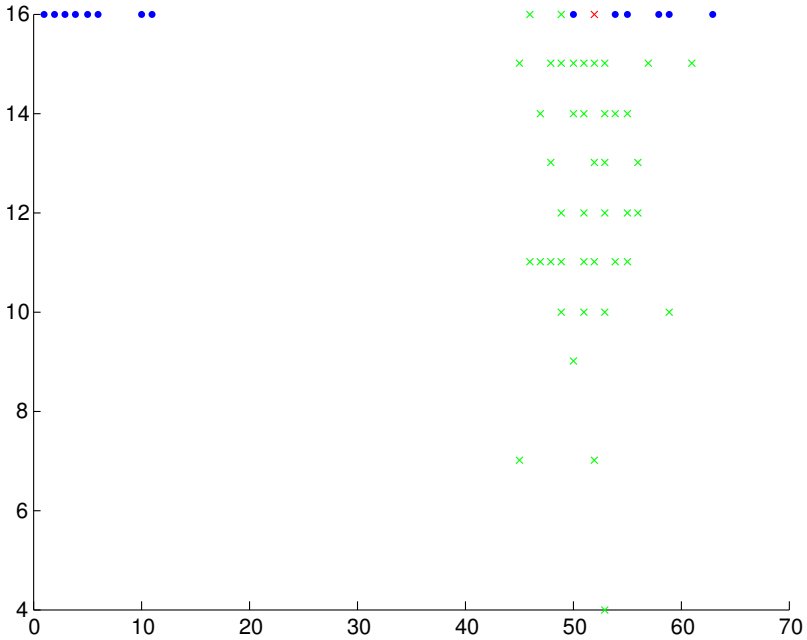


(b)

Abbildung 3: IE Verbindungen



(a)



(b)

Abbildung 4: II Verbindungen