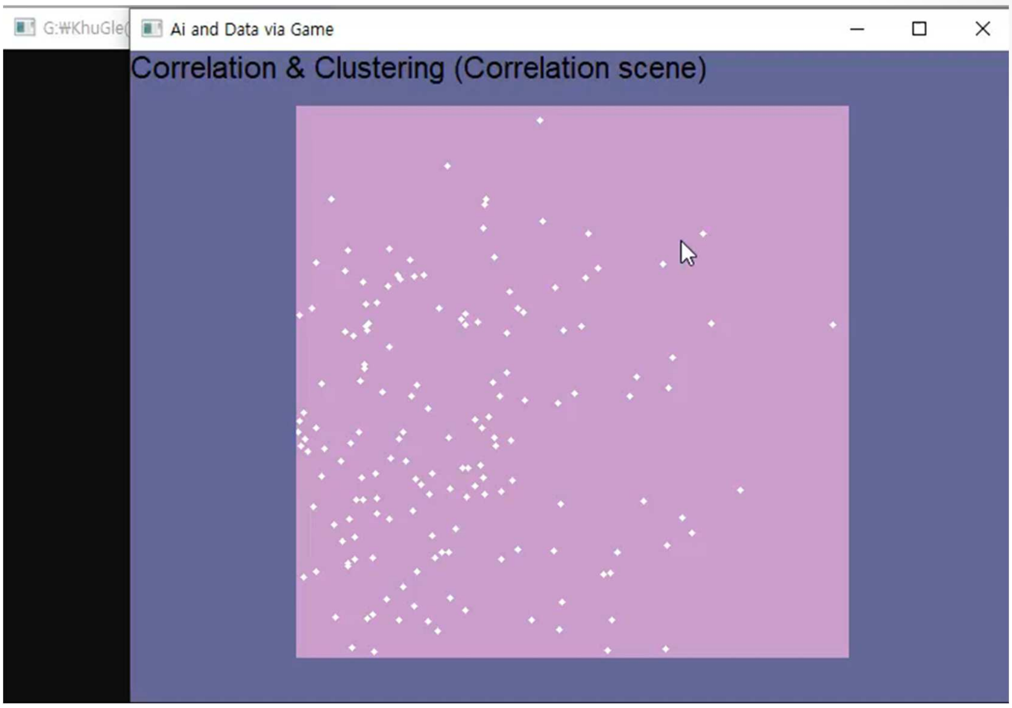
7. Correlation & Clustering



Pearson Correlation Coefficient

- Pearson correlation coefficient (PCC)
  - Pearson product-moment correlation coefficient

$$\rho_{x,y} = \frac{\mathrm{cov}(X,Y)}{\sigma_x \sigma_y} = \frac{\mathrm{E}\left[(X-\mu_x)(Y-\mu_y)\right]}{\sqrt{\mathrm{E}\left[(X-\mu_x)^2\right] E\left[(Y-\mu_y)^2\right]}}$$

$$= \frac{\mathrm{E}[XY]-\mathrm{E}[X]\mathrm{E}[Y]}{\sqrt{\left(\mathrm{E}[X^2]-(\mathrm{E}[X])^2\right)\left(\mathrm{E}[Y^2]-(\mathrm{E}[Y])^2\right)}}$$

$$= \frac{\sum(x_i-\mu_x)(y_i-\mu_y)}{\sqrt{\sum(x_i-\mu_x)^2 \sum(y_i-\mu_y)^2}}$$

Pseudo Random Number

```
int rand();        // pseudo random number, [0, RAND_MAX]
void srand();      // random seed initialization
srand(time(0));


// <random>
unsigned int seed = (unsigned int)std::chrono::
        system_clock::now().time_since_epoch().count();
std::default_random_engine generator(seed);

std::uniform_real_distribution<double> uniform_dist(0, 1);
std::normal_distribution<double> normal_dist(0, 1);

double number1 = uniform_dist(generator);
double number2 = normal_dist(generator);
```

KhuGleBase.cpp (1)

```
double GetPearsonCoefficient(std::vector<std::pair<double, double>> Data) {
  double Mean1 = 0, Mean2 = 0, Mean12 = 0;
  double SquaredMean1 = 0, SquaredMean2 = 0;

  for(auto EachData : Data) {
    Mean1 += EachData.first;
    Mean2 += EachData.second;
    Mean12 += EachData.first*EachData.second;

    SquaredMean1 += EachData.first*EachData.first;
    SquaredMean2 += EachData.second*EachData.second;
  }
```


KhuGleBase.cpp (2)

```
  Mean1 /= Data.size();
  Mean2 /= Data.size();
  Mean12 /= Data.size();

  SquaredMean1 /= Data.size();
  SquaredMean2 /= Data.size();

  double sigma1 = sqrt(SquaredMean1-Mean1*Mean1);
  double sigma2 = sqrt(SquaredMean2-Mean2*Mean2);

  if(sigma1 == 0 || sigma2 == 0) return 0;

  return (Mean12 - Mean1*Mean2)/(sigma1*sigma2);
}
```

Main.cpp (1)

```
…
class CCorrelationLayer : public CKhuGleLayer {
public:
  std::vector<CKhuGleSprite *> m_Point;

  CCorrelationLayer(int nW, int nH, KgColor24 bgColor,
    CKgPoint ptPos = CKgPoint(0, 0)): CKhuGleLayer(nW, nH, bgColor, ptPos) {
    GenerateData(200);
  }
  void GenerateData(int nCnt);
};
void CCorrelationLayer::GenerateData(int nCnt) {
  unsigned int seed = (unsigned int)std::chrono::
                       system_clock::now().time_since_epoch().count();
  std::default_random_engine generator(seed);
  std::uniform_real_distribution<double> uniform_dist(0, 1);

  for(auto &Child : m_Children)
    delete Child;
  m_Children.clear();
  m_Point.clear();
```


Main.cpp (2)

```
  double mean1 = uniform_dist(generator);
  double mean2 = uniform_dist(generator);

  double sigma1 = uniform_dist(generator)/2.;
  double sigma2 = uniform_dist(generator)/2.;

  double rotate = uniform_dist(generator)*Pi;

  std::normal_distribution<double> normal_dist1(mean1, sigma1);
  std::normal_distribution<double> normal_dist2(mean2, sigma2);
```

Main.cpp (3)

```
  double x, y;
  for(int i = 0 ; i < nCnt ; i++) {
    double xx = normal_dist1(generator);
    double yy = normal_dist2(generator);

    x = (xx-mean1)*cos(rotate) - (yy-mean2)*sin(rotate) + mean1;
    y = (xx-mean1)*sin(rotate) + (yy-mean2)*cos(rotate) + mean2;

    x = (x*m_nW - m_nW/2)*0.6 + m_nW/2;
    y = (y*m_nH - m_nH/2)*0.6 + m_nH/2;

    CKhuGleSprite *Point = new CKhuGleSprite(GP_STYPE_ELLIPSE, GP_CTYPE_DYNAMIC,
      CKgLine(CKgPoint((int)x-2, (int)y-2), CKgPoint((int)x+2, (int)y+2)),
      KG_COLOR_24_RGB(255, 255, 255), true, 30);

    m_Point.push_back(Point);
    AddChild(Point);
  }
}
```


Main.cpp (4)

```
class CClusterLayer : public CKhuGleLayer {
  …
};

class CCorrelationClustering : public CKhuGleWin {
public:
  CKhuGleScene *m_pCorrelationScene;
  CKhuGleScene *m_pClusteringScene;

  CCorrelationLayer *m_pCorrelationLayer;
  CClusterLayer *m_pClusteringLayer;

  bool m_bCorrelationScene;

  CCorrelationClustering(int nW, int nH);
  virtual ~CCorrelationClustering() {
    m_pScene = nullptr;
    delete m_pCorrelationScene;
    delete m_pClusteringScene;
  }
  void Update();
};
```

Main.cpp (5)

```
CCorrelationClustering::CCorrelationClustering(int nW, int nH)
  : CKhuGleWin(nW, nH) {
  m_pCorrelationScene = new CKhuGleScene(640, 480,
    KG_COLOR_24_RGB(100, 100, 150));
  m_pClusteringScene = new CKhuGleScene(640, 480,
    KG_COLOR_24_RGB(100, 100, 150));

  m_pCorrelationLayer = new CCorrelationLayer(400, 400,
    KG_COLOR_24_RGB(200, 150, 200), CKgPoint(120, 40));
  m_pCorrelationScene->AddChild(m_pCorrelationLayer);

  m_pClusteringLayer = new CClusterLayer(400, 400,
    KG_COLOR_24_RGB(150, 150, 200), CKgPoint(120, 40));
  m_pClusteringScene->AddChild(m_pClusteringLayer);

  m_pScene = m_pCorrelationScene;
  m_bCorrelationScene = true;
}
```


Main.cpp (6)

```
void CCorrelationClustering::Update() {
  if(m_bKeyPressed['M']) {
    m_bCorrelationScene = !m_bCorrelationScene;
    if(m_bCorrelationScene)
      m_pScene = m_pCorrelationScene;
    else
      m_pScene = m_pClusteringScene;
    m_bKeyPressed['M'] = false;
  }
  if(m_bKeyPressed['S']) {
    if(m_bCorrelationScene) {
      std::vector<std::pair<double, double>> Data;
      for(auto Point : m_pCorrelationLayer->m_Point)
        Data.push_back({Point->m_Center.x, Point->m_Center.y});

      double pcc = GetPearsonCoefficient(Data);
      std::cout << pcc << std::endl;
    }
    else {
    …
    }
    m_bKeyPressed['S'] = false;
  }
```

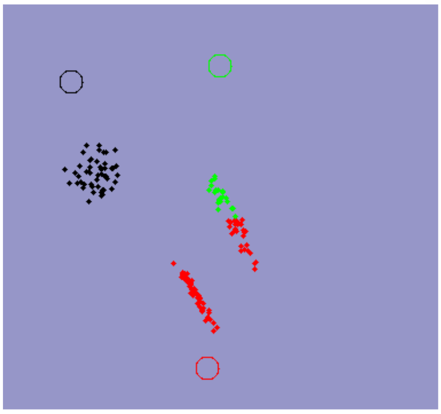
Main.cpp (7)

```
  if(m_bKeyPressed['N']) {
    if(m_bCorrelationScene)
      m_pCorrelationLayer->GenerateData(200);
    else {
    …
    }
    m_bKeyPressed['N'] = false;
  }
  m_pScene->Render();
  if(m_bCorrelationScene)
    DrawSceneTextPos("Correlation && Clustering (Correlation scene)", CKgPoint(0, 0));
  else
    DrawSceneTextPos("Correlation && Clustering (Clustering scene)", CKgPoint(0, 0));
  CKhuGleWin::Update();
}
int main() {
  CCorrelationClustering *pCorrelationClustering
    = new CCorrelationClustering(640, 480);
  KhuGleWinInit(pCorrelationClustering);
  return 0;
}
```


k-Means Clustering (1)

- k-Means Clustering
  - Centroid-based method
  - Iterative refinement method

  - $\mathbf{c}_0=\{c_1, c_2, \ldots c_k\} \leftarrow$ random
    while iteration or **c** is not change ($\mathbf{c}_i = \mathbf{c}_{i-1}$) **do**
    assign each sample to the cluster which has the closest **c**
    compute new centroids ($\mathbf{c}_i$) for each cluster (sample mean)
    end while



k-Means Clustering (2)

- Static k value
- Dependent results on initial centroids
- Spherical and equally sized clustering

Main.cpp (1)

```
…
class CKhuGleSprite2 : public CKhuGleSprite {
public:
  int m_nClusterIndex;
  CKhuGleSprite2(int nType, int nCollisionType, CKgLine lnLine,
    KgColor24 fgColor, bool bFill, int nSliceOrWidth = 100,
    int nClusterIndex = 0)
  : CKhuGleSprite(nType, nCollisionType, lnLine, fgColor, bFill, nSliceOrWidth)
  {
    m_nClusterIndex = nClusterIndex;
  }
};
```

Main.cpp (2)

```
class CClusterLayer : public CKhuGleLayer {
public:
  std::vector<CKhuGleSprite2 *> m_Center;
  std::vector<CKhuGleSprite2 *> m_Point;
  int m_nClusterNum, m_nStep;

  CClusterLayer(int nW, int nH, KgColor24 bgColor,
    CKgPoint ptPos = CKgPoint(0, 0)) : CKhuGleLayer(nW, nH, bgColor, ptPos) {
    m_nClusterNum = 3;

    GenerateData(m_nClusterNum, 50);
    m_nStep = 0;
  }

  void GenerateData(int nCluster, int nCnt);
};
```


Main.cpp (3)

```
void CClusterLayer::GenerateData(int nCluster, int nCnt) {
  unsigned int seed = (unsigned int)std::chrono
    ::system_clock::now().time_since_epoch().count();
  std::default_random_engine generator(seed);

  std::uniform_real_distribution<double> uniform_dist(0, 1);

  for(auto &Child : m_Children)
    delete Child;

  m_Children.clear();
  m_Center.clear();
  m_Point.clear();

  for(int i = 0 ; i < m_nClusterNum ; ++i) {
    CKhuGleSprite2 *Center = new CKhuGleSprite2(GP_STYPE_ELLIPSE,
      GP_CTYPE_DYNAMIC, CKgLine(CKgPoint(m_nW/2-10, m_nH/2-10),
      CKgPoint(m_nW/2+10, m_nH/2+10)),
      KG_COLOR_24_RGB(i%2*255, i/2%2*255, i/4%2*255), false, 100);

    m_Center.push_back(Center);
    AddChild(Center);
  }
```

Main.cpp (4)

```
for(int k = 0 ; k < nCluster ; ++k) {
  double mean1 = uniform_dist(generator);
  double mean2 = uniform_dist(generator);

  double sigma1 = uniform_dist(generator)/10.;
  double sigma2 = uniform_dist(generator)/10.;

  double rotate = uniform_dist(generator)*Pi;

  std::normal_distribution<double> normal_dist1(mean1, sigma1);
  std::normal_distribution<double> normal_dist2(mean2, sigma2);
```


Main.cpp (5)

```
    double x, y;
    for(int i = 0 ; i < nCnt ; i++) {
      double xx = normal_dist1(generator);
      double yy = normal_dist2(generator);

      x = (xx-mean1)*cos(rotate) - (yy-mean2)*sin(rotate) + mean1;
      y = (xx-mean1)*sin(rotate) + (yy-mean2)*cos(rotate) + mean2;

      x = (x*m_nW - m_nW/2)*0.6 + m_nW/2;
      y = (y*m_nH - m_nH/2)*0.6 + m_nH/2;

      CKhuGleSprite2 *Point = new CKhuGleSprite2(GP_STYPE_ELLIPSE,
        GP_CTYPE_DYNAMIC,
        CKgLine(CKgPoint((int)x-2, (int)y-2), CKgPoint((int)x+2, (int)y+2)),
        KG_COLOR_24_RGB(255, 255, 255), true, 30);

      m_Point.push_back(Point);
      AddChild(Point);
    }
  }
}
```

Main.cpp (6)

```
…
void CCorrelationClustering::Update() {
  …
  if(m_bKeyPressed['S']) {
    if(m_bCorrelationScene) {
      …
    }
  else {
    if(m_pClusteringLayer->m_nStep == 0) {
      for(auto &Center : m_pClusteringLayer->m_Center)
        Center->MoveTo((double)rand()/RAND_MAX*m_pClusteringLayer->m_nW,
          (double)rand()/RAND_MAX*m_pClusteringLayer->m_nH);
    }
    else {
      std::vector<int> ClusterCnt;
      std::vector<std::pair<double, double>> NewCenter;

      for(auto &Center : m_pClusteringLayer->m_Center) {
        NewCenter.push_back({0., 0.});
        ClusterCnt.push_back(0);
      }
```


Main.cpp (7)

```
      for(auto &Point : m_pClusteringLayer->m_Point) {
        int Index = Point->m_nClusterIndex;

        NewCenter[Index].first += Point->m_Center.x;
        NewCenter[Index].second += Point->m_Center.y;

        ClusterCnt[Index]++;
      }
      for(int k = 0 ; k < m_pClusteringLayer->m_nClusterNum ; ++k) {
        if(ClusterCnt[k] > 0)
          m_pClusteringLayer->m_Center[k]->MoveTo
            (NewCenter[k].first/ClusterCnt[k],
             NewCenter[k].second/ClusterCnt[k]);
      }
    }
```

Main.cpp (8)

```
    for(auto &Point : m_pClusteringLayer->m_Point) {
      double MinDist, Dist;
      for(int k = 0 ; k < m_pClusteringLayer->m_nClusterNum ; ++k) {
        Dist = sqrt((Point->m_Center.x
            - m_pClusteringLayer->m_Center[k]->m_Center.x)
           *(Point->m_Center.x - m_pClusteringLayer->m_Center[k]->m_Center.x) +
           (Point->m_Center.y - m_pClusteringLayer->m_Center[k]->m_Center.y)
           *(Point->m_Center.y - m_pClusteringLayer->m_Center[k]->m_Center.y));

        if(k == 0) {
          Point->m_nClusterIndex = k;
          MinDist = Dist;
        }
        else if(Dist < MinDist) {
          Point->m_nClusterIndex = k;
          MinDist = Dist;
        }
      }
      Point->m_fgColor = KG_COLOR_24_RGB(Point->m_nClusterIndex%2*255,
        Point->m_nClusterIndex/2%2*255, Point->m_nClusterIndex/4%2*255);
    }
    ++(m_pClusteringLayer->m_nStep);
    }
    m_bKeyPressed['S'] = false;
  }
```

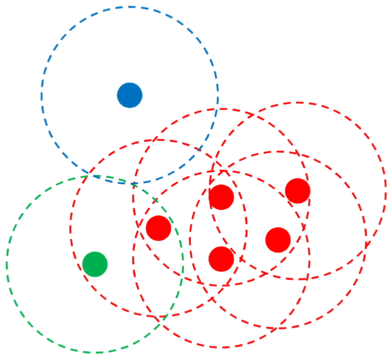

Main.cpp (9)

```
  if(m_bKeyPressed['N']) {
    if(m_bCorrelationScene)
      m_pCorrelationLayer->GenerateData(200);
    else {
      m_pClusteringLayer->GenerateData(m_pClusteringLayer->m_nClusterNum, 50);
      m_pClusteringLayer->m_nStep = 0;
    }
    m_bKeyPressed['N'] = false;
  }

  m_pScene->Render();
  …
}
int main() {
  …
}
```

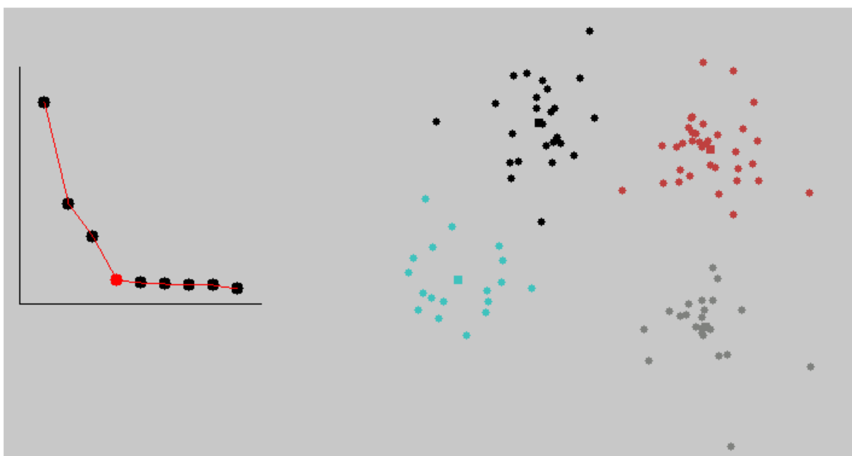
Practice VI (1)

- DBSCAN
  - Density-based spatial clustering of application with noise
  - Density-based clustering
  - Core points: at least τ points with distance ε
  - Border points: reachable from a core points
  - Outliers: not core points and not reachable form any core points



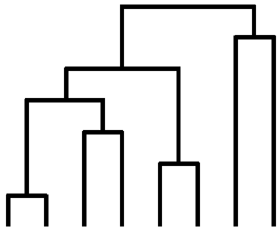
Practice VI (2)

- Elbow method
  - Determining the number of clusters
  - # of clusters vs. sum of squared distance
  - SSD (sum of squared distance, sse: sum of squared error)
    - Sum of squared distance from the cluster centroid



Advanced Courses (1)

- Connectivity-based clustering
  - Merge for split



- Clustering evaluation
  - Known class labels
    - Precision
    - RI (rand index)
  - Unknown class labels
    - Sum of squared distance
    - Silhouette value

Advanced Courses (2)

- Cross correlation
  - Similarity, matching score
  - Dot product

$$f(t) \otimes g(t) \triangleq \int_{-\infty}^{\infty} f^*(\tau) g(\tau + t) d\tau$$

$$f(t) * g(t) \triangleq \int_{-\infty}^{\infty} f(\tau) g(t - \tau) d\tau$$

$$\rho_{x,y} = \frac{\operatorname{cov}(X,Y)}{\sigma_x \sigma_y} = \frac{\mathrm{E}\left[\left(X - \mu_x\right)\left(Y - \mu_y\right)\right]}{\sigma_x \sigma_y} = \frac{1}{N} \frac{\left(X - \mu_x\right)\left(Y - \mu_y\right)}{\sigma_x \sigma_y}$$

- Cosine similarity

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos\theta$$

$$\cos\theta = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|}$$