Simple numerical techniques for mesoscale polymer models
Koopman, E.A.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 10

Appendix

10.1 Efficient clustering in unstructured networks

Here we list a piece of C++ code that can efficiently group together a set of unstructured points. We only know the point coordinates and we have a set of point index pairs representing links. The code refers in many places to the color of a point. Initially this is set to a unique index, each greater than the total number of particles, afterwards all points connected together share the same color. The code has been slightly rewritten to increase readability, for example the std::map⟨T,U⟩::find() == std::map⟨T,U⟩::end() construction might be unclear to people not familiar with C++ so this has been changed to psuedocode "contains()".

The algorithm works by effectively keeping track of two arrays of colors. The first is on the points themselves, the second is the colormap. The values in the first array are pointers into the second array. Each point in the second array points to the left or nowhere. If it points to the left, that indirection should be followed recursively until it points nowhere. If a point in the second array points nowhere, that index in the array is the final cluster index.

The first stage constructs the two arrays, the second stage performs the final cluster index lookup for each point.

```cpp
void Detector::ColorLoop(vector<CPoint> Points, vector<CLink> Links)
{
    map<int, int> ColorMap;
    map<int, int> ColorMapMap;
    int ClusterCount = 1;

    // Do a single pass over all links, construct a map that contains
    // redirections to eventually the correct cluster index.
    for (vector<CLink>::iterator iLink = Links.begin();
        iLink != Links.end(); ++iLink)
    {
```
int Color1 = m_Points[iLink->P1()].GetColor();
int Color2 = m_Points[iLink->P2()].GetColor();
int MaxColor = Color1 > Color2 ? Color1 : Color2;

if (Color1 < Color2)
    m_Points[iLink->P1()].SetColor(Color2);
else
    m_Points[iLink->P2()].SetColor(Color1);

// None found
if (!ColorMap.contains(Color1) && !ColorMap.contains(Color2))
{
    ColorMap[MaxColor] = ClusterCount;
    ClusterCount++;
}

// Only Color2 found
if (!ColorMap.contains(Color1) && ColorMap.contains(Color2))
{
    if (Color1 > Color2)
        ColorMap[Color1] = ColorMap[Color2];
}

// Only Color1 found
if (ColorMap.contains(Color1) && !ColorMap.contains(Color2))
{
    if (Color2 > Color1)
        ColorMap[Color2] = ColorMap[Color1];
}

// Both found, conflict, resolve by going all the way down
// and merging to lowest value
if (ColorMap.contains(Color1) && ColorMap.contains(Color2))
{
    int Lowest1 = ColorMap[Color1];

    while (ColorMap.contains(Lowest1) && (Lowest1 != ColorMap[Lowest1]))
        Lowest1 = ColorMap[Lowest1];

    int Lowest2 = ColorMap[Color2];

    while (ColorMap.contains(Lowest2) && (Lowest2 != ColorMap[Lowest2]))
        Lowest2 = ColorMap[Lowest2];

    if (Lowest1 < Lowest2)
        ColorMap[Lowest2] = Lowest1;
    else
        ColorMap[Lowest1] = Lowest2;
}

// For each point hop through the map to get to the actual index
// for its cluster
for (vector<CPnt>::iterator iPoint = m_Points.begin();
10.2 Structure factor calculation

Calculating the static structure factor can be a time consuming task if not done properly. The structure factor is given by

$$S(\vec{k}) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{N} e^{i(\vec{k} \cdot \vec{r}_{ij})}.$$ 

A careful observer will notice the double summation, something that you typically try to avoid in computer programs since it is an indication of at least $O(n^2)$ complexity.

Fortunately there is a neat way of rewriting the equation so that it becomes linear in the amount of particles. We start by writing $S(\vec{k})$ as

$$S(\vec{k}) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{N} N e^{i(\vec{k} \cdot \vec{r}_{ij})}.$$ 

$$= \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{N} e^{i(\vec{k} \cdot \vec{r}_{ij})} e^{i(\vec{k} \cdot \vec{r}_{ij})}.$$ 

$$= \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{N} e^{i(\vec{k} \cdot \vec{r}_{ij})} N e^{i(\vec{k} \cdot \vec{r}_{ij})}.$$ 

$$= \frac{1}{N} \sum_{i=1}^{N} \cos(\vec{k} \cdot \vec{r}_{ij}) + i \sin(\vec{k} \cdot \vec{r}_{ij}) \sum_{j=1}^{N} \cos(\vec{k} \cdot \vec{r}_{ij}) + i \sin(\vec{k} \cdot \vec{r}_{ij})$$ 

$$= \frac{1}{N} \sum_{i=1}^{N} \cos(\vec{k} \cdot \vec{r}_{ij}) + i \sin(\vec{k} \cdot \vec{r}_{ij}) \sum_{j=1}^{N} \cos(\vec{k} \cdot \vec{r}_{ij}) - i \sin(\vec{k} \cdot \vec{r}_{ij})$$
and now since all the \( \sin(\cdot) \cos(\cdot) \) terms vanish because they directly cancel each other out and all the \( \sin(\vec{k} \cdot \vec{r}_i) \cos(\vec{k} \cdot \vec{r}_j) \) with \( i \neq j \) are on average 0 we get

\[
S(\vec{k}) = \frac{1}{N} \sum_{i=1}^{N} \cos^2(\vec{k} \cdot \vec{r}_i) + \sin^2(\vec{k} \cdot \vec{r}_i) \quad (10.6)
\]

which is a single summation.

Now the second thing to keep in mind when calculating structure factors is that the length associated with \( \vec{k} \) has to fit an integer number of times in the system box. An abbreviated version of our actual implementation is shown here. This code is easily extended to handle multi species structure factors.

```c
void StructureFactor(int Mode, SParameters Prmtrs, SParticle* pPrts)
{
  static double Sk[3][SIZE]; // SIZE = 100
  static int updates = 0;
  double Kv[3][SIZE];
  switch(Mode)
  {
    case MODE_INIT:
    {
      for (int i = 0; i < SIZE; i++)
      {
        Sk[0][i] = Sk[1][i] = Sk[2][i] = 0;
        for (int dim = 0; dim < 3; dim++)
          Kv[dim][i] = (double)(i + 1) * (2 * PI / Prmtrs.boxsize[dim]);
      }
      break;
    }
    case MODE_UPDATE:
    {
      updates++;
      for (int dim = 0; dim < 3; dim++)
      {
        for (int ki = 0; ki < SIZE; ki++)
        {
          double a = 0, b = 0;
          for (int i = 0; i < Prmtrs.NP; i++)
          {
            a += cos (Kv[dim][ki] * pPrts[i].Pos[dim]);
            b += sin (Kv[dim][ki] * pPrts[i].Pos[dim]);
          }
          Sk[dim][ki] += a*a + b*b;
        }
      }
    }
  }
}
```
break;

case MODE_FINALIZE:
    for (int i = 0; i < SIZE; i++)
        for (int dim = 0; dim < 3; dim++)
            Sk[sim][i] /= (double)(updates * Prmtrs.NP);
    // Sk now contains correct structure factor
    break;
}