Parallel programming project: Exploring the Strauss point process model

Linda Stougaard Nielsen
Laboratory for Computational Stochastics
University of Aarhus

Abstract

With point of reference in a theoretical problem, parallel programming is discussed. The theoretical problem is to describe the mean number of points and neighbours in a Strauss point process as a function of the three parameters. The problem is embarrassing parallel. Emphasis is given on describing the construction of a parallel program and explaining the code. Monte Carlo methods are used for handling point processes. Some technical details are given.

1. Introduction

Recently the Laboratory for Computational Stochastics bought 16 PC's with each 2 CPU's. By connecting the computers we have a so-called Beowulf cluster consisting of 32 CPU's. Our cluster was named *Beostar*. Such a cluster can be used for parallel programming.

In order to get some hand-on experience using Beostar, we launched a project using an all-ready existing `C++` library for handling point processes. The objective was then to write the surrounding `MPI` code to administer the parallel environment (Message Passing Interface, see Snir et al. (1996)).

This report describes the project and the theoretical results. Emphasis is given on the technical part describing the parallel program thoroughly and the sequential program in some detail. The sequential program enters as an isolated part of the parallel program and the parallel program can therefore with a few changes be modified for solving other problems which are embarrassing parallel.

All programs, sequential as well as parallel, were developed by the writer and are available for scientific purposes on `www.imf.au.dk/stoclab` and come with following disclaimer: All use at own risk – no guarantees given.

In Section 2 the background for the project is found. We give a short introduction to point processes, in particular the Strauss point process, which is the model to be studied. We want to make a table for the mean number of points and mean number of neighbours for varying parameters. In Section 3 a sequential program for doing this is presented. In Section 4 some initial

investigations of the sequential program is discussed before we in Section 5 discuss how the sequential program can be parallelised. The parallel program is given together with technical explanations of the standard `MPI` procedures used. In Section 6 the results of the investigation is discussed. Issues of future interest are discussed in Section 7.

In Appendix A we find some background on Monte Carlo techniques used for simulating in point process models, including an example of the Metropolis-Hastings birth-death algorithm. In Appendix B the *loess* algorithm for local estimation of surfaces is given. Appendix C gives some practical instructions for using Beostar. In Appendix D we find a result concerning scaling of point processes.

2. Background: Point processes

Data sets consisting of positions of events or objects are often observed in nature. Examples are the positions of detected α–particles generated by decay of Radon gases (Stoyan et al., 1995, Figure 2.5), stain centres in methylene-blue coloured clay observed in a horizontal slice (Lieshout, 2000, Figure 1.1), position of cells in tissue (Ripley, 1977, Figure 2) and (Nielsen, 2000, Figure 1), position of trees in a forest (Diggle, 1983, Figure 1), (Ripley, 1977, Figure 10), (Ogata and Tanemura, 1986, Figure 1), and (Stoyan and Stoyan, 1998, Figure 1), or bronze particles in a filter (Hahn et al., 2001, Figure 1). Notice that all these examples are planar point patterns. Point patterns may also be observed in 1 or 3 dimensions or on the surface of a body. Data might even be of higher dimensions if e.g. the point pattern consists of vectors of various data as in multivariate statistics. However, typically the points represent positions.

Point process models are stochastic models for point patterns. The point process model chosen to describe a given point pattern must be able to describe the characteristics of the pattern. There are two main types of characteristics: Homogeneity and interaction. A point pattern is homogeneous if the distribution of the points is translation invariant meaning that the point pattern on average looks similar in different areas. This implies e.g. that the mean number of points in an area is the same if the area is translated. In this report we only consider homogeneous point patterns. If a point pattern is not homogeneous we call it inhomogeneous. For more details on inhomogeneous point process models, the reader may consult e.g. Jensen and Nielsen (2001) or Nielsen (2001) and references therein.

Interaction is when the positions of points are not independent. Thus, the position of one point depends on the positions of the other points in the realization. There are two main types of interaction: Clustering and inhibi-

tion. Clustering is when the point process model favour point patterns where the points lie in clusters. Thus, there are many small distances between the points. Inhibition (regularity) is when the point process model favour point patterns where the points do not lie close to one another. There are few small distances between the points in the patterns. By considering distances between points in the pattern to either other points in the pattern or to arbitrarily chosen points, it is possible to make a non-parametric distinction between clustering and inhibition where the homogeneous Poisson point process serve as the division between the two types of interaction. See Diggle (1983), Stoyan et al. (1995), and Baddeley and van Lieshout (1995) for more details. The homogeneous Poisson point process model is also called the model of complete spatial randomness since all the points are independently and homogeneous distributed. Hence, this way we also get a non-parametric test of the hypothesis of complete spatial randomness.

Among the examples mentioned above, the point pattern in Ripley (1977, Figure 2) is clearly regular and homogeneous, see e.g. Baddeley and van Lieshout (1995) where the J function is plotted for these data. The point patterns in Stoyan et al. (1995, Figure 2.5) and Diggle (1983, Figure 1) are also homogeneous with a tendency of inhibition. However, a homogeneous Poisson point process might be sufficient in both cases. The latter point pattern has also been analysed in Baddeley and van Lieshout (1995). Examples of homogeneous clustered point patterns can be seen in Lieshout (2000, Figure 1.1) and Ripley (1977, Figure 10). The latter is also analysed in Baddeley and van Lieshout (1995). The point patterns in Stoyan and Stoyan (1998, Figure 1), Nielsen (2000, Figure 1), and Hahn et al. (2001, Figure 1) are regular and inhomogeneous. Finally, we see a clustered inhomogeneous point pattern in Ogata and Tanemura (1986, Figure 1). However, for this data set an inhomogeneous Poisson model might also be sufficient.

In this paper we will consider the Strauss point process which is the most commonly used point process model for point patterns showing inhibition. For other models and models describing clustering, good places to look is Baddeley and Møller (1989) and Geyer (1999). Notice that it is not always evident to differentiate between interaction and inhomogeneity. As an example, see Brix and Møller (1998), where clustering is modelled using an inhomogeneous Poisson point process with a stochastic intensity function.

2.1. The Strauss point process

Let $\mathcal{X} \subseteq \mathbb{R}^m$ be a full-dimensional bounded subset. In the rest of the paper $\mathcal{X} = [0;1] \times [0;1]$, the unit square. We will let X denote a point process on $\mathcal{X}$ and use x for a realization of the point process. Since realizations are

point patterns, the state space of X is $\Omega_{\mathcal{X}}$, which is the set of all finite point patterns on $\mathcal{X}$. The Greek letters η and ξ denotes single points.

A Strauss point process X on $\mathcal{X}$ (Strauss, 1975) has density (with respect to the unit rate Poisson point process) of the form

$$f(x) = c(\beta, \gamma, r)^{-1} \beta^{n(x)} \gamma^{s_r(x)}, \quad x \in \Omega_{\mathcal{X}}, \tag{1}$$

where $n(x)$ is the number of points in the point pattern x and

$$s_r(x) = \sum_{\{\eta,\xi\} \subseteq x} 1(\|\eta - \xi\| < r)$$

is the number of pairs of points in x with a distance smaller than r. Such pairs are also called neighbours, which comes from the fact that the Strauss process belongs to the class of Markov point process models, cf. Ripley and Kelly (1977). Thus, the density depends on the number of points and the number of neighbours in the realization. The model has three parameters

$$(\beta, \gamma, r) \in (0; \infty) \times (0; 1] \times [0; \infty).$$

From the density we see that γ controls the strength of the interaction, since the density is punished with a factor γ for each pair of neighbours in the point pattern x. Thus, the smaller the γ, the fewer neighbours we expect and the more inhibition there will be between the points in the realizations of the model. When $\gamma = 1$ we have the Poisson model (no interaction). The parameter r controls the range of interaction, where $r = 0$ is the Poisson model. The last parameter β controls the number of points. However, the situation is more complicated than this, see Figure 1 where realizations are shown for various values of the parameters β and γ. The number of points increases in β as well as in γ, and the number of neighbours also increases in β simply because the number of points increases. Furthermore, the number of points decreases and the and the number of neighbours increases if the interaction distance r is increased.

It is of interest to explore the dependence between the parameters and the expected number of points and neighbours, i.e. to describe the functions,

$$\begin{array}{rcl} (\beta, \gamma, r) & \to & \mathbb{E}_{\beta,\gamma,r}\, n(X), \\ (\beta, \gamma, r) & \to & \mathbb{E}_{\beta,\gamma,r}\, s_r(X). \end{array} \tag{2}$$

For $\gamma = 1$ or $r = 0$ X is the Poisson model with intensity β and

$$\mathbb{E}_{\beta,\gamma,r}\, n(X) = \beta. \tag{3}$$

[Tilsvarende resultat for EsX (afh af beta og r)?] OBS

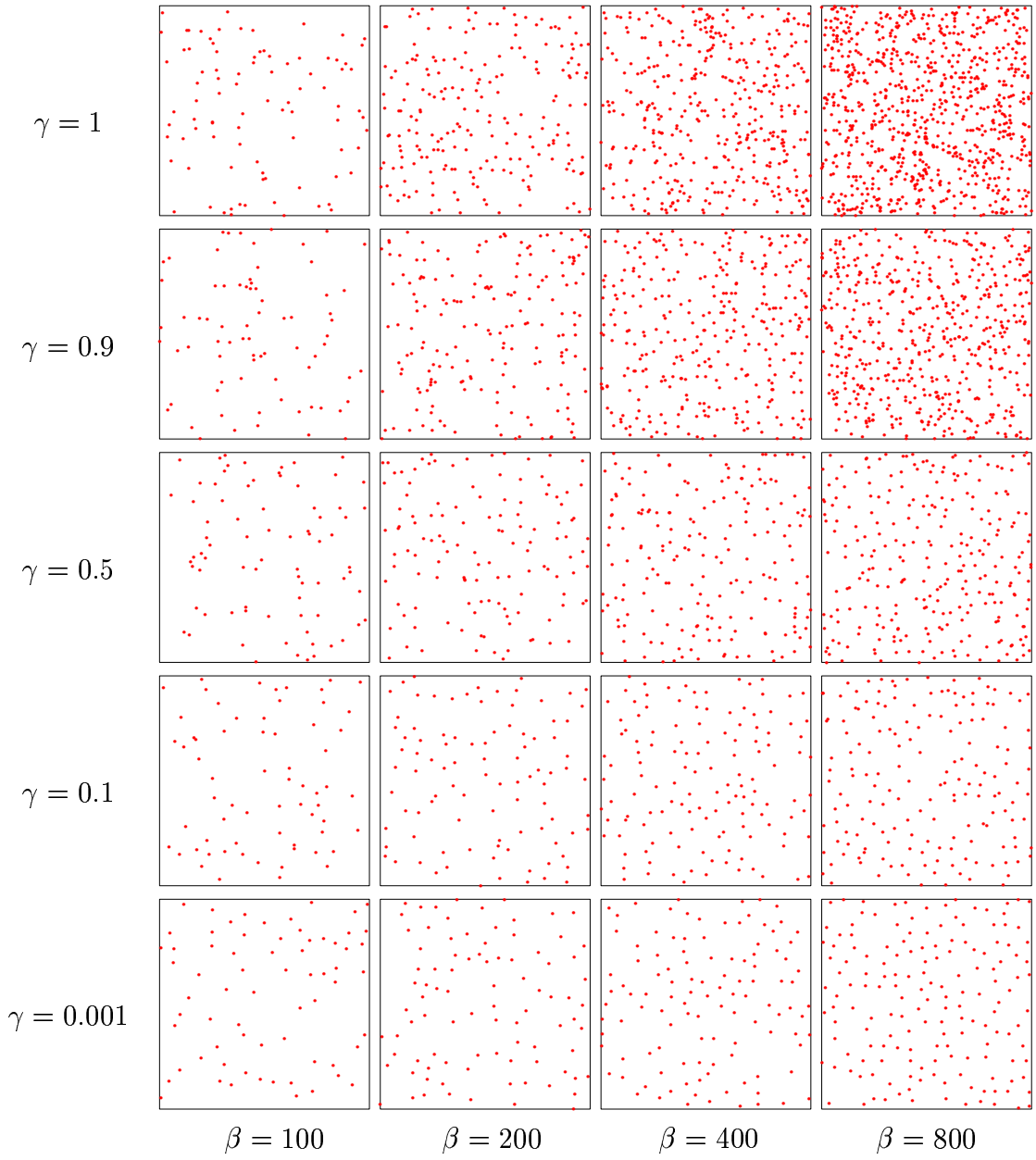


Figure 1: Strauss point processes on $\mathcal{X} = (0; 1)^2$ with fixed $r = 0.05$

For $\gamma \in (0;1)$ and $r > 0$ it is not possible to compute the means (2) in terms of the density function (1) since the constant $c(\beta, \gamma, r)$ is intractable. It is not even possible to approximate this constant. The constant depends on the parameters, so it is essential when we want to study the behaviour of the means as functions of the parameters. Instead we approximate the means for a given value of (β, γ, r) by simulating under the model and then approximate the means by sample means.

2.2. Maximum likelihood analysis

Apart from exploring the dependence of the parameters, there is also another reason that we want to tabulate the means of the number of points and neighbours in the Strauss model.

When r is fixed the density (1) is of exponential family form. It can be shown, that the maximum likelihood estimate of (β, γ) based on an observed point pattern x, is then the unique solution to the equations

$$\mathbb{E}_{\beta,\gamma,r}\, n(X) = n(x) \quad \text{and} \quad \mathbb{E}_{\beta,\gamma,r}\, s_r(X) = s_r(x), \tag{4}$$

see e.g. Geyer (1999). Thus, suppose we have a table

$$(\beta, \gamma, r) \to (\mathbb{E}_{\beta,\gamma,r}\, n(X), \mathbb{E}_{\beta,\gamma,r}\, s_r(X)) \tag{5}$$

for all values of β, γ, and r. Then given r, we can find the unique entry where

$$(\mathbb{E}_{\beta,\gamma,r}\, n(X), \mathbb{E}_{\beta,\gamma,r}\, s_r(X)) = (n(x), s_r(x)),$$

which gives us the maximum likelihood estimate of β and γ.

It is interesting to investigate how much these estimates depends on the choice of r. By varying r, we find a function

$$r \to (n(x), s_r(x)) \to (\hat{\beta}(r), \hat{\gamma}(r)). \tag{6}$$

If r is not known, the maximum likelihood estimates are the value of r, and the corresponding values (6), which maximizes the partially maximized likelihood function,

$$\overline{L}(r; x) = \max_{\beta,\gamma} L(\beta, \gamma, r; x) = L(\hat{\beta}(r), \hat{\gamma}(r), r; x).$$

Here we gain from having tabulated the function (6).

Thus, an important aspect of doing likelihood inference in the Strauss process, no matter if r is known or not, is to find the values of (β, γ) that solves the equations (4). To do this we have to simulate for various values of (β, γ) until the simulated sample means of the number of points and

neighbours are close enough to the observed values. Hence, by tabulating these means once and for all, we can save this time consuming step of the estimation procedure. Furthermore we get the opportunity to explore how the means depend on the parameters and how, and more importantly, how much the maximum likelihood estimates of (β, γ) changes for varying r.

3. Sequential program

In this section we consider a sequential program that computes the functions (2). We can only get point-wise approximations of the means, and therefore we make a grid of (β, γ, r) values and determine the values of the means in each grid point. Such a grid has of course to be chosen large enough to cover all realistic point patterns, see Figure 1, and fine enough such that the values between the grid points can be realistic approximated. We will return to this in Section 4. Given a grid has been constructed, the sequential program for tabulating the functions (2) in the grid points takes form as sketched in Figure 2. For each value of (β, γ, r), (2) are approximated as sample means over `nsamp` realizations from the Strauss point process with parameters (β, γ, r). This is what happens in the lines 4–12 in the program. In the lines 6–8, one realization from the Strauss process is simulated by running `nburn` steps in a Metropolis-Hastings Birth-Death (MHBD) algorithm starting with the empty point configuration. See Appendix A.1 for further explanations, where also the algorithm for a single step in a MHBD algorithm is given (line 8).

```
for ( (β, γ, r) ∈ grid )
  {
    initialize (β, γ, r)
    for ( i = 0; i < nsamp; i++ )
      {
        x_i = ∅
        for ( j = 0; j < nburn; j++ )
            run x_i 1 step in MHBD algorithm
        save n(x_i) and s_r(x_i)
      }
    E_{β,γ,r} n(X) = (1/nsamp) Σ_{i=1}^{nsamp} n(x_i)
    E_{β,γ,r} s_r(X) = (1/nsamp) Σ_{i=1}^{nsamp} s_r(x_i)
  }
```

Figure 2: Sequential program for computing the means (2).

3.1. First time saving operation

The `nsamp` samples used to approximate the means have to be independent of each other, hence, the procedure sketched in Figure 2, where we start simulating from scratch for each sample, seems to be the ideal solution. However, one of the real time consuming parts of this program is that we have to run `nburn` steps, which typically is a very large number, in the MHDB algorithm before we can be sure that we have reached equilibrium (see Appendix A.1). A very common way to obtain many samples from a distribution is to draw the samples from the same chain, cf. e.g. Geyer (1999). First we run a chain `nburn` steps in order to reach equilibrium. After that, we get pseudo-independent samples by resume with running the same chain and sample every `nspac`'th sample. This is illustrated in Figure 3. Letting `nburn`$= 50000$, `nsamp`$= 200$, and `nspac`$= 1000$, we need to go $2.5 \cdot 10^5$ steps in a MHDB algorithm. Using the option from Figure 2, the number of steps is 10^7. Thus, we save a factor 40 in time (approximately, since the first steps in a chain will run faster).

Between the grid points, we change the parameters and it is therefore not possible to sample from one chain for all the grid points. However, we could let the chain start in a neighbouring grid point with the same value of r. Then it would probably not take that long to reach equilibrium. This

```
for ( (β, γ, r) ∈ grid )
  {
    initialize (β, γ, r)
    x = ∅
    for ( i = 0; i < nburn; i++ )
      run x 1 step in MHBD algorithm
    for ( i = 0; i < nsamp; i++ )
      {
        for ( j = 0; j < nspac; j++ )
          run x 1 step in the same MHBD algorithm
        save n(x_i) = n(x) and s_r(x) = s_r(x_i)
      }
    E_{β,γ} n(X) = (1/nsamp) Σ_{i=1}^{nsamp} n(x_i)
    E_{β,γ} s_r(X) = (1/nsamp) Σ_{i=1}^{nsamp} s_r(x_i)
  }
```

Figure 3: Sequential program replacing the program in Figure 2. Here we sample using spacing.

is however not a large time saving factor. In the example above, the factor is at most 1.25. Considering the extra programming work and storage space involved, this is really not worth while.

3.2. Perfect simulation

In theory the chain is known to reach equilibrium after infinitely many steps. A problem with the above procedure is therefore that we can never be really sure that we have reached equilibrium. Furthermore, different burn-in times are most likely necessary for different parameter values. A way to solve this problem is to use perfect simulation, cf. **?**. In perfect simulation, the sample is drawn from time zero supposing that the chain was started in time minus infinity. Thus, in time zero we can be sure to have reached equilibrium and can start the sampling using spacing as before.

The trick using perfect simulation is that by constructing the algorithm in some smart way, we only need to look back finitely many steps because at a certain point, using the same random numbers, any point configuration will end up in the same point configuration at some time before zero (coalescence). Thus, we have but one point configuration at time zero which then is the perfect sample drawn from the chain in equilibrium.

Implementing perfect simulation would involve replacing the lines 5–6 in the program in Figure 3. Such an algorithm have at present not been implemented. Notice that the time saving factor is not large, if any.

4. Initial investigations of the sequential program

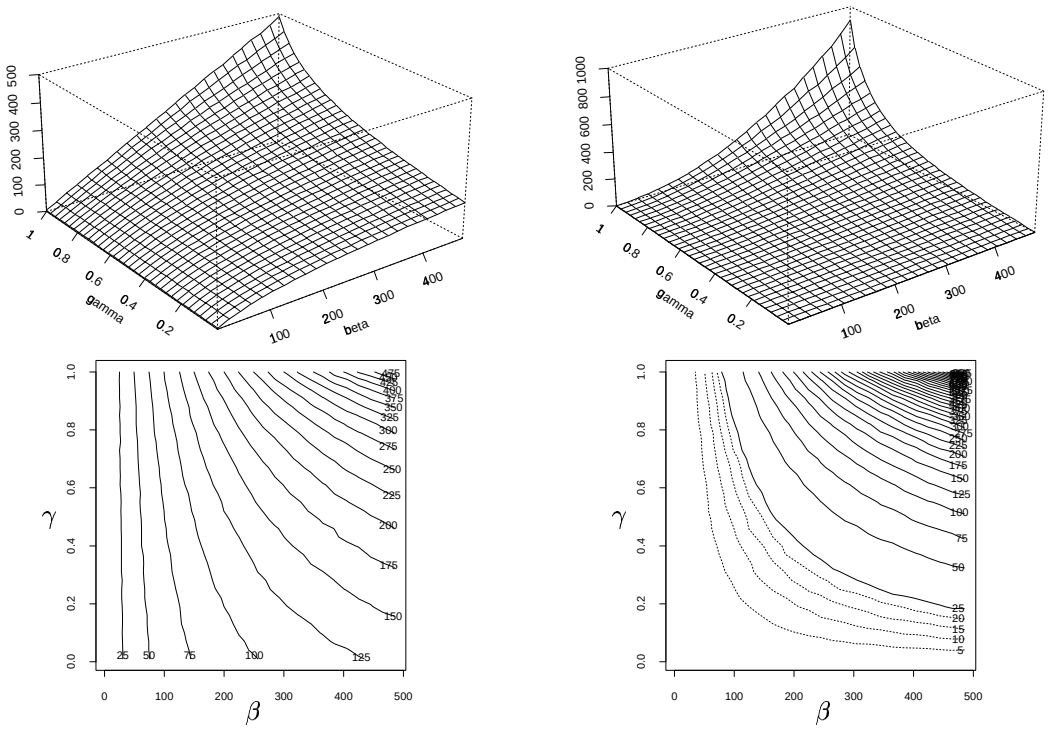
The functions (2) were tabulated for $r = 0.05$ fixed and for various values of (β, γ) in a grid.

In Figure 4 the simulated values of $\mathbb{E}_{\beta,\gamma,r}\, n(X)$ and $\mathbb{E}_{\beta,\gamma,r}\, s_r(X)$ are plotted against the grid point values of (β, γ). The data points are plotted as surfaces with lines connecting neighbouring grid points. The corresponding level plots are shown below.

First we observe that the theoretical result (3) holds. $\gamma = 1$ corresponds to the line hitting the back wall of the upper plot in Figure 4 (a). This line is approximately increasing with unit rate. Furthermore, the mean number of points increases in β and in γ as expected, and so does the mean number of neighbours.

In Figure 5 the point-wise difference between the means resulting from two separate runs with $r = 0.05$ have been plotted. The differences are not large relatively to the absolute values.

The fluctuations in the data surfaces are believed to be a consequence of



(a) $\mathbb{E}_{\beta,\gamma,r} n(X)$ (b) $\mathbb{E}_{\beta,\gamma,r} s_r(X)$

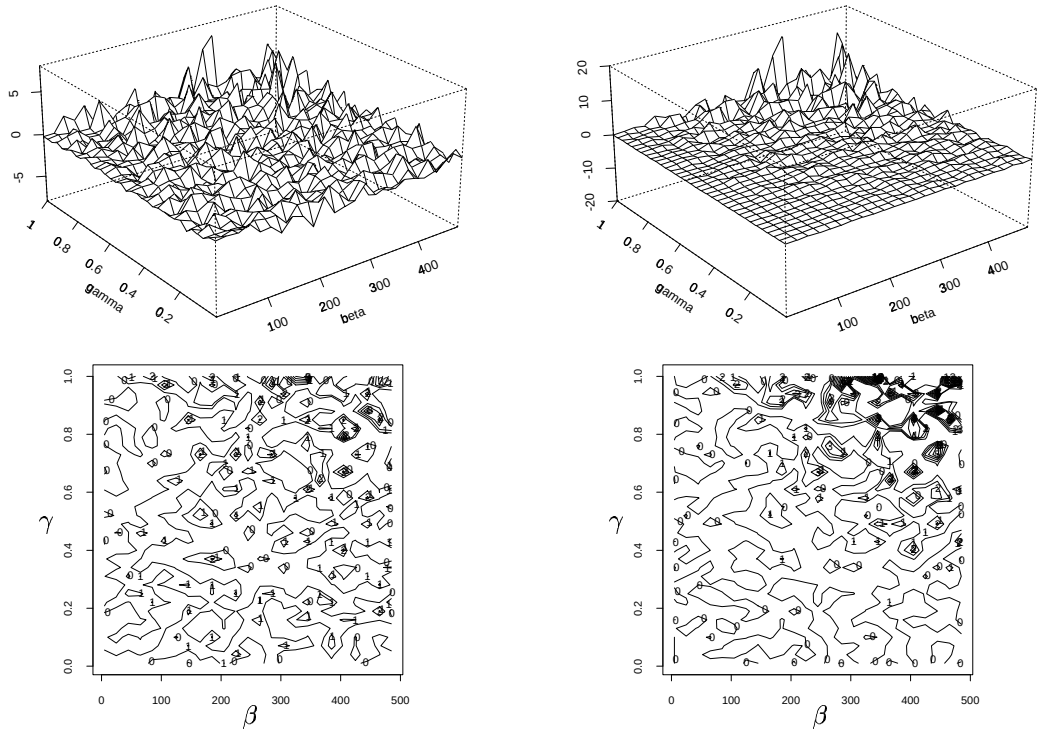
Figure 4: Point-wise plots of the functions (2) for fixed $r = 0.05$ with lines connecting grid point values. Lower plots are level plots.

the approximation. In Figure 6 the data have been smoothed using the local smoothing algorithm *loess* (local regression, see Appendix B). The difference between the smoothed surfaces and the data surfaces is approximately constant and 0, see Figure 7 where the differences have been plotted. No areas have larger fluctuation (relative to the absolute numbers).

Thus, since the fluctuations in the data plots are very small, we conclude that the chosen values of `nburn` $= 50000$, `nsamp` $= 200$, and `nspac` $= 1000$ in the MHDB algorithm are sufficiently large, and that the tabbing between the grid values used in Figure 4,

$$\beta \in \{5, 25, 45, 65, ..., 485\}$$
$$\gamma \in \{0.01, 0.04, 0.07, ..., 1.00\}$$

does not need to be closer. Many different grids of various sizes have been tried out and also non-regular grids where the grid point distances were smaller where the surfaces were expected to vary more. We concluded that the regular coarse grid used in Figure 4 produced nice and smooth surfaces.



(a) $\mathbb{E}_{\beta,\gamma,r} n(X)$ residuals

(b) $\mathbb{E}_{\beta,\gamma,r} s_r(X)$ residuals

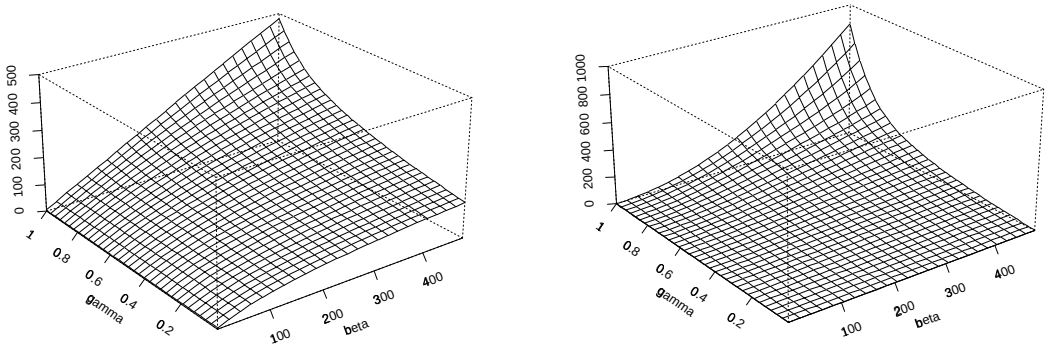
Figure 5: Point-wise differences between data from two separate runs computing (2) with $r = 0.05$ fixed. The two runs were from the sequential and parallel program, respectively.

However, the upper bound of β need to be increased to get point patterns with more points when the interaction is strong, see Figure 1.

For exploring the surfaces, it is sufficient to consider the data plots as in Figure 4. We do not gain much information about the behaviour by smoothing. However, for maximum likelihood estimation of (β, γ, r) in the Strauss point process, we use the smoothed surfaces in Figure 6 since we believe that the true mean surfaces are smooth and we therefore expect to get a better results by smoothing.

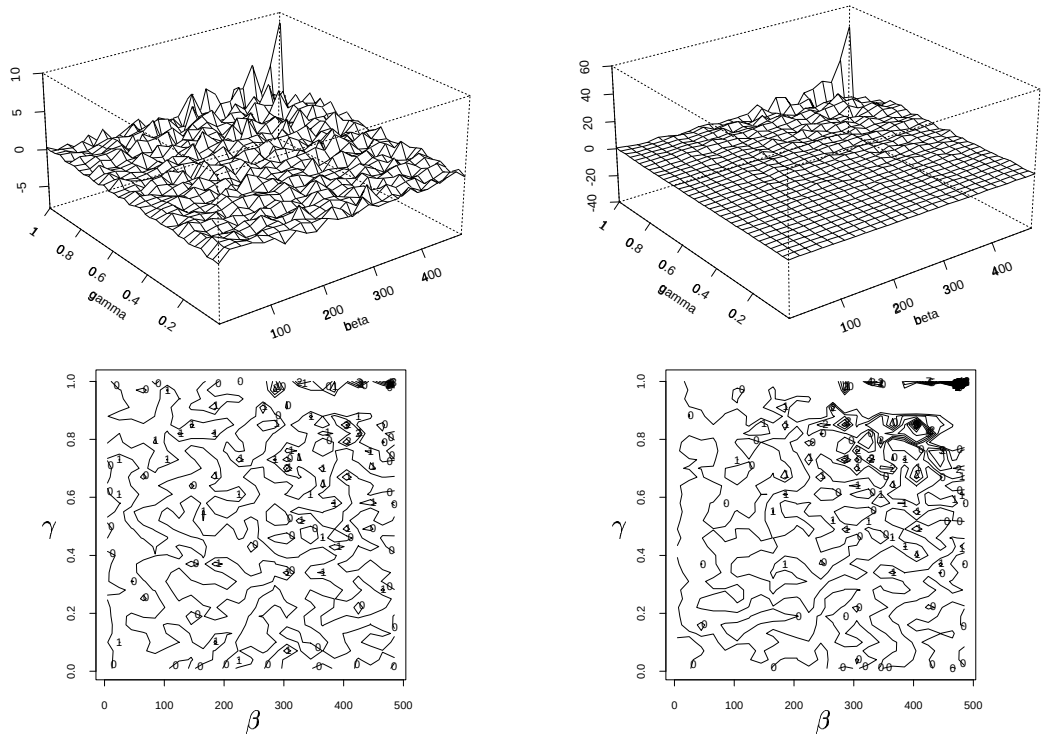
5. Embarrassing parallel program

Suppose we have a problem that can be split up into several minor problems, henceforth called *jobs*, which can be solved independently of each other and in any order. By running the jobs on n different CPU's, we in principle get a factor n speed-up. A program that administer the allocation of the jobs



(a) $\mathbb{E}_{\beta,\gamma,r} n(X)$ (b) $\mathbb{E}_{\beta,\gamma,r} s_r(X)$

Figure 6: Estimated surfaces based on data from Figure 4 using the *loess* smoothing algorithm.



(a) $\mathbb{E}_{\beta,\gamma,r} n(X)$ residuals (b) $\mathbb{E}_{\beta,\gamma,r} s_r(X)$ residuals

Figure 7: Point-wise differences between data from Figure 4 and the fitted surfaces from Figure 6.

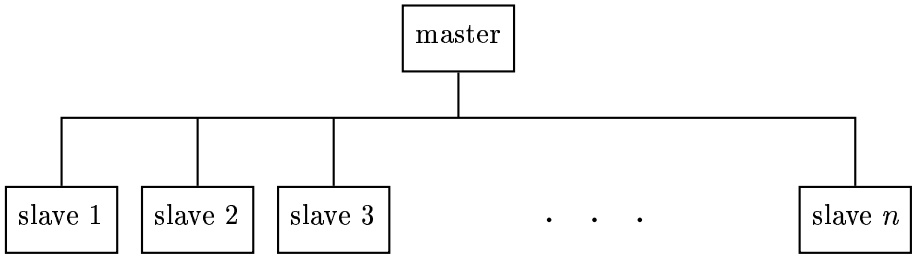


Figure 8: The communication structure in an embarrassing parallel program.

on different CPU's and collect the results, is an embarrassing (or perfect) parallel program.

Such a program can be run on a Beowulf cluster computer and typically has the following structure. One CPU will be reserved to administer the allocation of jobs and to collect all the results. This is henceforth called the *master* CPU. The other CPU's, called *slaves*, do the actual work. Communication is between master and the slaves only. There is no communication between the slaves. The master gives a slave a job which the slave solves and report back to the master and gets a new job until all the jobs have been solved. See illustration in Figure 8.

Notice that parallel programming can also be used when the minor problems depend on each other and exchange of information is needed between the slaves. This is however much more complicated. See Pancake (1996) for more information on parallel programming in general.

5.1. Parallelising for point processes

As mentioned in Section 2.1, the density of a point process is in general intractable and therefore we need to use iterative methods such as Monte Carlo techniques when working with point processes, see Appendix A where these methods are briefly introduced. In general, Monte Carlo methods involves drawing independent realizations from the same distribution and from the distribution with various parameter values. Thus, the same operation, drawing a realization, is performed over and over.

A natural thought is therefore to let one job consist of drawing one sample. However, as discussed in Section 3.1, we can save a lot of computing time by drawing samples from the same Markov chain. In the particular example, we saved a factor 40. By parallelising where a job is drawing one sample, we save a factor 31 on Beostar (32 CPU's where one is the master). Thus, even though a Beowulf cluster provides us with a huge amount of computer power, it is important still to optimize the sequential algorithm before beginning to parallelise. If the optimization blocks for the parallelising, one must consider

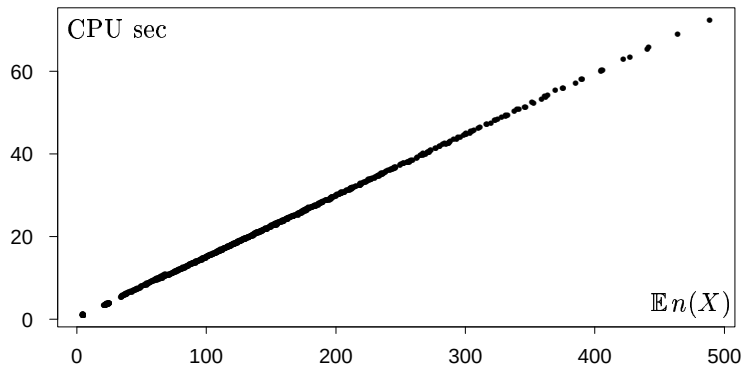


Figure 9: The CPU time is plotted against the simulated expected number of points (the sample average). The parameter $r = 0.05$ is fixed and (β, γ) varies.

which solution gives the best speed-up.

5.2. Parallelising the current problem

To solve the current problem, to tabulate the functions (2) for different values of (β, γ, r), we first draw `nsamp` realizations from the same distribution in order to form the means for one value of (β, γ, r). This is done for each value of (β, γ, r) in the grid. As discussed above, the part where the realizations are draw from the same distribution can be speeded up by sampling from the same Markov chain. Since the steps in the chain subsequently depend on each other, parallelising these steps is not feasible. Instead we parallelise the grid points. Thus, one job consist of computing the means for a given value of (β, γ, r). This is the content of the outer loop (lines 3–14) of the sequential program in Figure 3. For simplicity, this job is made into a single procedure

```
estimate_means( nburn, nsamp, nspac, EnX, EsX)
```

in the `C++` library handling point processes. The resulting means are put into the last two entries.

In Section 4 we studied the sequential program for $r = 0.05$ fixed. The CPU times were measured for each job. The time is linearly dependent on the number of points in the patterns, see Figure 9 where the simulated expected number of points is plotted against the CPU time. This comes as no surprise since the work that has to be done (count the number of neighbours to one point) is proportional to the number of points in the given point configuration.

Thus, the time it takes for one jobs takes various time. Recall that the number of points increases in γ and β and decreases in r and therefore the

durance of a job behaves similarly. In order to avoid that we in the end only have one job left which takes longest time, and therefore have only one single CPU working in the end, we will treat the jobs in order approximately decreasing in time. Thus, we start with the grid point with largest β and γ and smallest r and end with the grid point with smallest β and γ and largest r.

Notice that the reusing of point patterns in the outer loop, as discussed in the end of Section 3.1, is much less feasible to implement now, since a neighbouring grid point point pattern will often not be available and furthermore we would have to use huge amount of storage room to store all these point patterns. This requires much more coordination and some slaves might have to wait in order to get a point pattern from a close enough other grid point. The algorithm would become very complicated, which is base for more mistakes and much programming time.

5.3. The parallel program with comments

In the following we run through the parallel program corresponding to the sequentail program in Figure 3. The program is written in `C++` code but will in principle look similar in `C` or `FORTRAN`. The choise of language is only important for the surrounding code. The code controlling the parallelising enters as isolated procedures and some data types, and can be recognised since they all start with the letters `MPI` referring to Message Passing Interface. The `MPI` code is identical in `C` and `C++` and slightly different in `FORTRAN`. See Snir et al. (1996) for the exact syntax. This is also a good reference for other procedures. We will in the following only use the most simple, which are described below.

The structure of the parallel program is sketched in Figure 10. It starts with a preamble including libraries and defining global variables. The user-constructed library `ppunfix5.h` deals with point processes (among other it contains the function `estimate_means`). The three *tags* defined in the preamble will be used in the communication between the master and the slaves. The integer `number_of_processors` will later be set to contain the number of CPU's working.

After the preamble follows all the procedures. In this case there are three carrying the names `master`, `slave`, and `main`. A `C++` program must always contain a `main` procedure. This procedure is always run, whereas other procedures are only run if called. Below I will give a thorough description of each of these procedures.

When running a parallel program then the same program, the one in Figure 10, is run on each of the CPU's in the Beowulf cluster. Recall that

```
#include <mpi.h>
#include "ppunfix5.h"
#include ... other libraries ...

#define WORKTAG 1
#define READYTAG 2
#define DIETAG 3

int number_of_processors;

void master ( void )
{
  ... code ...
}

void slave ( void )
{
  ... code ...
}

main( int argc, char **argv )
{
  ... code ...
}
```

Figure 10: A sketch of the embarrassing parallel program.

the idea in an embarrassing parallel program is that one CPU works as master and the others as slaves. Therefore we have a `master` and a `slave` procedure doing these two different tasks, and basically the only thing happening in the `main` procedure (which is run by every CPU) is that one CPU is told to run the `master` procedure and all the other CPU's are told to run the `slave` procedure.

The `main` procedure is shown in Figure 11. Each CPU has an identity number (on Beostar from 0 to 31). This number is read into the local variable `myrank` by the call `MPI_Comm_rank` and is used to determine which one of the two procedures `master` and `slave` is to be run by the current CPU. Only the CPU with identity number 0 runs the `master` procedure and all the rest run the `slave` procedure.

The function `MPI_init` passes information from the system call of the program. In `MPI_Size` sets the global parameter `number_of_processors` to

```
main( int argc, char **argv )
{
  int myrank;

  MPI_init( &argc, &argv );
  MPI_Size( MPI_COMM_WORLD, &number_of_processors );
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );

  if ( myrank == 0 )
    master();
  else
    slave();

  MPI_Finalize();
}
```

Figure 11: The `main` procedure of the parallel program from Figure 10.

the number of processors (CPU's). Beostar has 32.

In the end, when the current CPU has finished running either the master or the slave procedure, it returns to the `main` procedure and the whole program is shut down by calling `MPI_Finalize`.

Notice that this is the typical structure of an embarrassing parallel program.

Two central `MPI` *procedures*

In the following we run through the `master` and the `slave` procedures. First we consider the two most central commands in the communication, a send and a receive function,

```
MPI_Send( BUF, COUNT, DATATYPE, DEST,   TAG, COMM, IEERROR )
MPI_Recv( BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS  )
```

The three first entries describes what is sent/received and where it can be found. `COUNT` (integer) entries of type `DATATYPE` are sent/received and can be found in the memory from place `BUF` which is a pointer. `DATATYPE` is a `MPI` datatype, see Snir et al. (1996, p. 19), we are going to use `MPI_DOUBLE` which is a real number. Basically, what we put at the place `BUF` is the name of a vector of length `COUNT` and type that corresponds to `DATATYPE` (i.e. `DOUBLE` in our case).

`DEST` and `SOURCE` (integers) are the source and the destination of the message (identity number of the CPU).

`TAG` (integer) is a tag used in the communication. To make the program easier to read, we have defined some tag names as aliases for integers in the preamble, see Figure 10.

`COMM` is a communicator ... OBS

...`IEERROR` ... OBS

The variable `STATUS` of type `MPI_Status` records various status variables. Some functions come with this type. The ones we are going to use are `MPI_SOURCE` which returns the identity number of the CPU from whom the message was received, and `MPI_TAG` which returns the tag that was received.

The master procedure

Thus, one CPU (the one with identity number 0) is the master and the other ones are slaves working for the master. The purpose of the master is to allocate the work and the purpose of the slaves is to do the actual work. The master communicates with the slaves, telling the slaves what to do and collects the results of the work. It keeps track on which work in the stack of work (in this case a three dimensional grid of (β, γ, r) values) that have been performed and which still needs to be done. Finally, the master will write all the results on a file.

The `master` procedure is given in Figure 12. Below follow comments to the code where numbers in `< >` are used as references.

`<1>` First the grid is initialized. The start and end-values and step-size between grid points are given for each of the parameters β, γ and r. Then the integer `gridsize` is set to be the total number of grid points.

`<2>` A matrix named `alldata` of dimensions `gridsize`$\times 6$ and a vector named `gridorder` of length `gridsize` are created. Each row of `alldata` is of the form

$$\texttt{alldata[i] = (i, } \beta\texttt{, } \gamma\texttt{, } r\texttt{, } \mathbb{E}_{\beta,\gamma,r}\, n(X)\texttt{ , } \mathbb{E}_{\beta,\gamma,r}\, s_r(X)\texttt{)}$$

The first four places are initialized at present time and later the slaves will compute the last two values. The first column contains the index number. This is such that we quickly can find the right row again when a slave returns a vector with a done job.

The vector `gridorder` contains the integers `1,...,gridsize`. It controls the order in which to allocate the jobs of `alldata`, see Section 5.2. Thus, the `j`'th job to be allocated is `alldata[gridorder[j]]`.

`<3>` The variable `status` of type `MPI_Status` was explained above. The vector `package` is a temporary container for information received from the slaves. It is also used as dummy message from the master to the slaves. The integer `gridcount` keeps track on how far in the grid we have come

and `number_of_alive_nodes` is the number of slaves that are still working, which is initialized to be the number of CPU's minus 1 (the master).

`<4>` First we send a dummy message to each slave just telling the slaves to return the tag `READYTAG`.

`<5>` Then we run through the grid and assign jobs to the slaves until there is no more work. A slave is killed and `number_of_alive_nodes` is counted 1 down when there are no more grid points.

`<6>` In each loop we first receive something from the slave.

`<7>` If the master receives a `WORKTAG`, then the slave has dealt with one grid point and the result, $\mathbb{E}_{\beta,\gamma,r}\, n(X)$ and $\mathbb{E}_{\beta,\gamma,r}\, s_r(X)$, is put into the right place in the `alldata` matrix. Note that if we receive a `READYTAG` then the slave has not yet performed any work. The last tag, `DIETAG`, is never received.

`<8>` If there are no more undone grid points, then the waiting slave is told to terminate and we count the number of alive slaves one down.

`<9>` Else, we send the next job in the stack of work, the grid point values in `alldata[gridorder[gridcount]]`, to the slave and count the grid-counter one up. Note that we send work regardless of which tag we have received from the slave. There are three tags `READYTAG`, `WORKTAG` and `DIETAG`. The latter is never received but only send to the slaves ordering it to terminate. Receiving the first two, then the slave is ready to do another job.

```
void master ( void )
{
  ... code: initialize grid <1> ...

  double alldata[gridsize][6];
  int gridorder[gridsize];

  ... code: create grid (alldata and gridorder) <2> ...

  MPI_Status status;  // <3>
  double package[6];
  int gridcount = 0;
  int number_of_alive_nodes = number_of_processors - 1;

  for ( i = 1; i < number_of_processors; i++ )  // <4>
    MPI_Send( package, 6, MPI_DOUBLE, i, READYTAG, MPI_COMM_WORLD);
```

Figure 12: Continues on the following page. The `master` procedure of the parallel program in Figure 10.

```
  while ( number_of_alive_nodes > 0 ) // <5>
    {
      MPI_Recv( package, 6, MPI_DOUBLE, MPI_ANY_SOURCE,
                MPI_ANY_TAG, MPI_COMM_WORLD, &status); // <6>

      if ( status.MPI_TAG == WORKTAG )
        {
          ... code: save results from slave <7> ...
        }

      if ( gridcount >= gridsize )  // <8>
        {
          MPI_Send( package, 6, MPI_DOUBLE,  status.MPI_SOURCE,
                    DIETAG, MPI_COMM_WORLD);
          number_of_alive_nodes--;
        }
      else // <9>
        {
          MPI_Send( alldata[gridorder[gridcount]], 6, MPI_DOUBLE,
                    status.MPI_SOURCE, WORKTAG, MPI_COMM_WORLD);
          gridcount++;
        }
    }

    ... code: write results to file <10> ...

  return(); // <11>
}
```

Figure 12: Continued from previous page. The `master` procedure of the parallel program in Figure 10.

<10> Now, all grid points have been treated and we can save the results to a file.

<11> Finally, the master's job is done and the current CPU returns to the main program where it terminates.

The slave procedure
A slaves task is to do the jobs assigned by the master and report the results back. In this case a slave computes $\mathbb{E}_{\beta,\gamma,r}\, n(X)$ and $\mathbb{E}_{\beta,\gamma,r}\, s_r(X)$ given

a grid point (β, γ, r) from the master. Recall that this is the outcome of `estimate_means`.

The `slave` procedure is given in Figure 13 with comments following below.

`<12>` Initializing variables. `srand48(time(0))` initializes the randomness (ensures that the random numbers are indeed random). The three integers `nburn`, `nsamp`, and `nspac` are needed for the MHBD algorithm, see Appendix A.1. The variables `beta`, `gamma`, and `r` are in the beginning set to anything. The variable `pp` belongs to the class `straussprocess` which is defined in the library `ppunfix5.h` together with procedures doing various operations on a point process. For now it is just initialized to be of dimension 2 in the unit square (default) with the given parameters.

`<13>` See `<3>` above.

`<14>` The slave CPU starts to work until it is told to terminate.

`<15>` First we receive a message from master. Notice that we only receive from the master CPU which has identity number 0.

`<16>` If we have received the tag `DIETAG` from the master, then our task has ended and we return to the main part of the program, where the current CPU that was slave will terminate.

`<17>` If we have received the tag `READYTAG` from the master, then we do nothing but answer back with a dummy message and a `READYTAG`, see `<4>` and `<7>` above.

`<18>` In the last case, if we have received the tag `WORKTAG` from the master, then the master has sent a grid point which we have to work on.

`<19>` The first three functions initialize the parameters (β, γ, r) to be the values received from the master,

$$(\beta, \gamma, r) = (\texttt{ package[1], package[2], package[3] })$$

The means $\mathbb{E}_{\beta,\gamma,r} n(X)$ and $\mathbb{E}_{\beta,\gamma,r} s_r(X)$ are estimated by `estimate_means` and the results are put into the last two entries of the vector `package`.

Notice that these 4 lines, if surrounded by a loop running through the grid, basically constitutes the sequential program, see Figure 3. Thus, in order to solve another similar problem, these four lines is therefore (in principle) what needs to be modified. The rest of all the code just allocate the grid point values.

`<20>` Finally we send the result back to the master (CPU number 0) together with the tag `WORKTAG` telling the master "I have produced a result and am ready for more work".

```
void slave (void)
{
  srand48(time(0)); // <12>
  int nburn = 50000, nsamp = 200, nspac = 1000;
  double beta = 700, gamma = 0.1, r = 0.01;  // dummies
  straussprocess *pp;
  pp = new straussprocess(2,beta,gamma,r);

  MPI_Status status; // <13>
  double package[6];


  while ( true ) // <14>
    {
      MPI_Recv( package, 6, MPI_DOUBLE, 0,
                MPI_ANY_TAG, MPI_COMM_WORLD, &status ); // <15>

      if ( status.MPI_TAG == DIETAG ) // <16>
        return();

      if ( status.MPI_TAG == READYTAG ) // <17>
        MPI_Send(package, 6, MPI_DOUBLE, 0, READYTAG, MPI_COMM_WORLD);

      if ( status.MPI_TAG == WORKTAG ) // <18>
        {
          pp -> resetBeta(package[1]);  // <19>
          pp -> resetGamma(package[2]);
          pp -> initRelafst(package[3]);
          pp -> estimate_means(nburn, nsamp, nspac, package[4], package[5]);

          MPI_Send( package, 6, MPI_DOUBLE, 0,
                    WORKTAG, MPI_COMM_WORLD ); // <20>
        }
    }
}
```

Figure 13: The `slave` procedure of the parallel program in Figure 10.

6. Results of the investigations

In Figure 5 the difference between two runs with $r = 0.05$ fixed were plotted. These data were obtained from running the sequential program and the parallel program, respectively. Thus, as a first check, we observe that we obtain the same results as those from the sequential program (apart from random fluctuations).

The surfaces can be plotted for each value of r in the grid as those for $r = 0.05$ fixed in the upper plots of Figure 4. By running these plots as an animation with r as the time, we get moving surfaces. We observe that the movements are smooth. In Figure 14 the loess estimated surfaces (c.f. Appendix B) of the means have been plotted for the r values 0.03, 0.04, 0.05, 0.06, and 0.07.

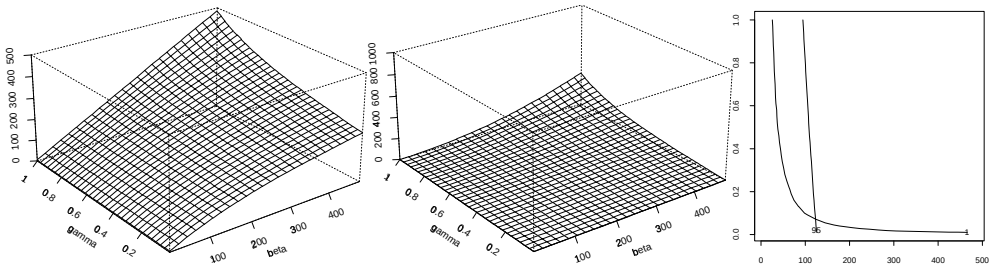
Both means increase in β and in γ. For r increasing, the number of neighbours explode for both β large and γ large. This is expected since given the number of points, then the larger the r, the more neighbours there are simply because it is more likely that two points are neighbours. In the extreme case where $r > \sqrt{2}$ then all points are neighbours, $s(x) = \frac{1}{2}n(x)(n(x)-1)$. Thus, the number of neighbours is quadratic in the number of points.

For $r = 0$ and for $\gamma = 1$ we have the Poisson process and $\mathbb{E}_{\beta,\gamma,r}\, n(X) = \beta$, c.f. (3). Thus, the backline of the plot showing the mean number of points is always fixed as the straight line. For $r = 0$ the surface is the plane $(\beta, \gamma) \to \beta$ and for r increasing, the plot falls down for small γ values and large β values. This is because there is very strong inhibition if γ small and the model therefore favours point patterns where very few pairs of points are closer than r apart. Thus, when r increases, there is less and less room for the points and there will therefore be fewer points. The effects of γ small implying $s_r(x)$ small dominates the effect of β large implying $n(x)$ large. This phenomenon is stronger the smaller the γ. And therefore we also see that the neighbour mean plot is still close to 0 for small γ values even when r is very large. The model just fix the 'problem' with a large r by having few points in the realizations.

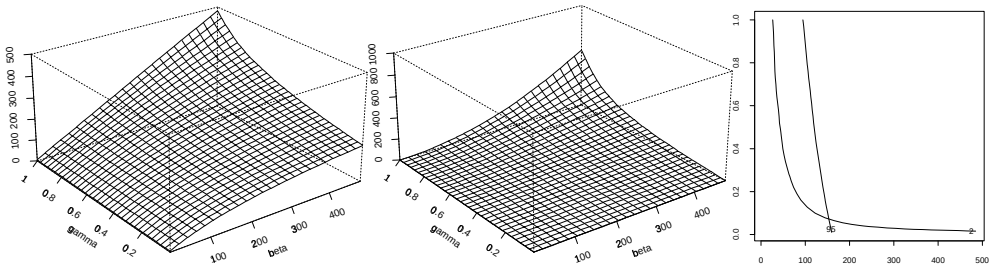
6.1. Simulation experiment

The best way to check the result is to make a simulation experiment and check that the parameters put into the experiment are close to those obtained by using the tabulated data for maximum likelihood estimation of the parameters.

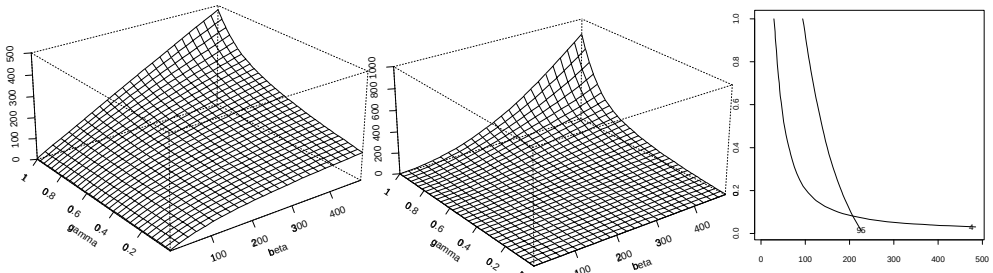
The simulated data is the Strauss point pattern from Figure 1 with parameters $(\beta, \gamma, r) = (200, 0.1, 0.05)$. We want to estimate the parameters (β, γ) and do this by solving the equations (4). The number of points in the



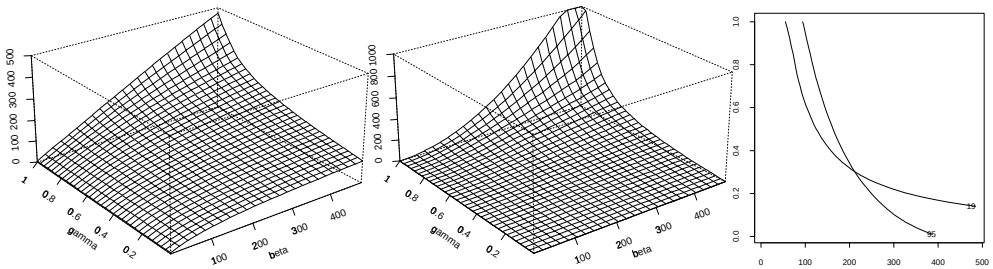
$r = 0.03$, $s_r(x) = 1$, and $(\hat{\beta}(r), \hat{\gamma}(r)) = (122, 0.075)$



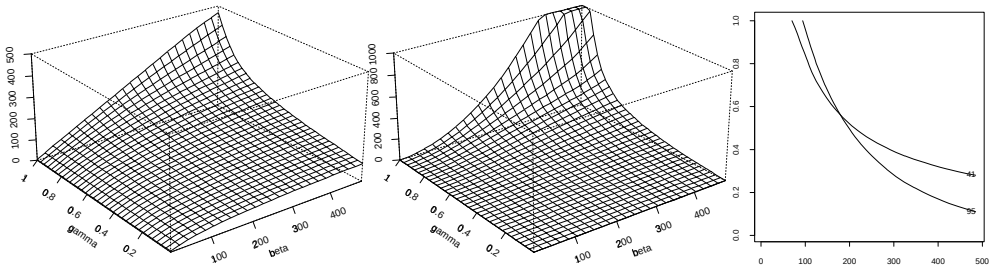
$r = 0.04$, $s_r(x) = 2$, and $(\hat{\beta}(r), \hat{\gamma}(r)) = (155, 0.075)$



$r = 0.05$, $s_r(x) = 4$, and $(\hat{\beta}(r), \hat{\gamma}(r)) = (211, 0.08)$



$r = 0.06$, $s_r(x) = 19$, and $(\hat{\beta}(r), \hat{\gamma}(r)) = (211, 0.31)$



$r = 0.07$, $s_r(x) = 41$, and $(\hat{\beta}(r), \hat{\gamma}(r)) = (178, 0.56)$

Figure 14: Simulation experiment for computing maximum likelihood estimates of simulated data using tabulated means. Loess surface for $\mathbb{E}_{\beta,\gamma,r} n(X)$, $\mathbb{E}_{\beta,\gamma,r} s_r(X)$, and level curves corresponding to the observed values of $n(x)$ and $s_r(x)$.

pattern is $n(x) = 95$. The number of neighbours depends on r. In Figure 14 we have plotted the loess estimated surfaces for the means for the r values 0.03, 0.04, 0.05, 0.06, and 0.07. Furthermore, the two level curves corresponding to the observed values of $n(x)$ and $s_r(x)$ are plotted. These level curves are thereby the solution sets to the two equations (4) and the point where the curves intersect solves both equations and is thereby the maximum likelihood estimate $(\hat{\beta}(r), \hat{\gamma}(r))$. As an example, for $r = 0.03$ we have $(n(x), s_r(x)) = (95, 1)$ and then the level curve for $\mathbb{E}_{\beta,\gamma,r} n(X) = 95$ and the level curve $\mathbb{E}_{\beta,\gamma,r} s_r(X) = 1$ are plotted. The curves intersect in $(122, 0.075)$.

It is interesting to notice that the maximum likelihood estimate is close to the true value when $r = 0.05$, the true value. Thereby it seems that the tabulated data are correct – at least in this particular area.

Furthermore, we see that the maximum likelihood estimate does not change very much for $r < 0.05$ but changes drastically for $r > 0.05$. This is of course due to the fact that the number of neighbours is very small and have a very limited range when making r smaller, whereas it can increase explosively for r larger. It could be interesting to make similar experiments for other point patterns where the true value of γ is larger such that there is more room for neighbours for values above the true r as well as for values below.

7. Discussion

7.1. The case where $\mathcal{X}$ is not the unit square

In this work we have restricted attention to the case where $\mathcal{X}$ is the unit square. The mean functions (**??**) have been tabulated when X is a Strauss point process in the unit square. If then instead $\mathcal{X} = (0; a) \times (0; b)$, $a, b > 0$, is a product set of the plane which is not necessarily the unit square, do we then have to tabulate a new table in order to find the means? The answer is no. We have following result which is a direct consequence of the transformation result for point processes (Jensen and Nielsen, 2000, Proposition 3.2), see Appendix D.

Corollary 1 *Let X be a Strauss point process on $\mathcal{X} = (0; 1) \times (0; 1)$ with parameters (β, γ, r). Then $Y = aX$ is a Strauss point process on $(0; a) \times (0; a)$ with parameters $(\frac{\beta}{a^2}, \gamma, ar)$.*

Thereby the means behave similarly for different sets. Furthermore maximum likelihood estimation can be based on the results from the table based on the unit square. Thus, suppose that we have observed a point pattern in the set $(0; a) \times (0; a)$. Then we scale the point pattern with a factor a

and estimate the parameters in the scaled point pattern x/a. The maximum likelihood of the parameters based on the original observed point pattern x is then

$$(\hat{\beta}, \hat{\gamma}, \hat{r}) = \left(\frac{\hat{\beta}_0}{a^2}, \hat{\gamma}_0, a\hat{r}_0 \right)$$

where $(\hat{\beta}_0, \hat{\gamma}_0, \hat{r}_0)$ is the maximum likelihood estimate based on x/a.

Notice that when scaling the point pattern in the unit square with a factor a then all the distances between the points are scaled with a factor a, thus, it is intuitively evident that also the value of r is scaled with a factor a. Likewise, we can think of β controlling the intensity, the number of points per unit area. Thus, scaling the point pattern means that the number of points per unit area is a factor $1/a^2$ smaller since the total area is a^2 larger. The parameter γ controls the strength of the interaction which is not changed by the scaling. Such considerations are the main concepts behind the scaling approach presented in Hahn et al. (2001).

Suppose now that we have observed a point pattern x in $(0; a) \times (0; b)$. Then scale x with a factor $c = \sqrt{ab}$ such that the area of the scaled set is 1. Again we find the maximum likelihood estimate $(\hat{\beta}_0, \hat{\gamma}_0, \hat{r}_0)$ based on the scaled point pattern $x/\sqrt{ab}$, and since the area is 1 and $n(x)$ and $s_r(x)$ depend on the relative size but not dimensions (disregard edge effects) then we can use the results for the unit square. The maximum likelihood estimates based on x is then

$$(\hat{\beta}, \hat{\gamma}, \hat{r}) = \left(\frac{\hat{\beta}_0}{ab}, \hat{\gamma}_0, \sqrt{ab}\hat{r}_0 \right).$$

7.2. Future work

As discussed in Section 6.1, further studies about the maximum likelihood estimation could be interesting. Perfect simulation could be implemented in order to obtain better sampling as discussed in Section 3.1.

It is also interesting to find out whether the model can be reparametrized such that fewer parameters are sufficient. Ute Hahn suggests to use $(\beta r^2, \gamma)$.

Other issues of interest are to do similar investigations for other interaction models and models allowing for inhomogeneity.

Furthermore we need to incorporate edge effects since the true number of neighbours might also depend on the points outside the observation window.

Acknowledgements

This project was supported .... MaPhySto ... Laboratory ... OBS

Apart from the writer, following persons, to whom I am very grateful for their contributions, help and support, were involved in the project: Peter Møller-Nielsen (Computer Science Department, University of Aarhus), Michael Kjærgaard Sørensen, and Eva B. Vedel Jensen (both from Laboratory of Computational Stochastics).

A. Simulating in point process models

The density of a point process is intractable. Following Geyer (1999), let $h_\psi(x)$ be the explicitly known part of the density parametrized by $\psi \in \Psi$, then the density is of the form

$$f_\psi(x) = \frac{h_\psi(x)}{c(\psi)},$$

where

$$c(\psi) = \int_{\Omega_\mathcal{X}} h_\psi(x)\Pi(dx)$$

is an integral over all possible point patterns with respect to the probability measure of the unit rate Poisson point process. This integral can be expressed as an infinite sum of multiple integrals over $\mathcal{X}$ with respect to the usual Lebesgue measure

$$\int_{\Omega_\mathcal{X}} h_\psi(x)\Pi(dx) = \sum_{n=0}^{\infty} \mathrm{e}^{-\mathrm{area}(\mathcal{X})} \frac{1}{n!} \int_\mathcal{X} \cdots \int_\mathcal{X} h_\psi(\{x_1, \ldots, x_n\}) dx_1 \cdots dx_n.$$

However, it can still not be computed explicitly or even approximated. The way we deal with point processes is then to base everything in interest on ratios of densities.

There are now two possibilities. Either the parameter is the same or it is not. In the first case,

$$\frac{f_\psi(x)}{f_\psi(y)} = \frac{h_\psi(x)}{h_\psi(y)},$$

and the ratio is explicitly expressed. Such ratios are used when we want to simulate a realization from the model with parameter ψ. Then we construct a Markov chain and build the transition probabilities on such ratios, see Appendix A.1 where the Metropolis-Hastings birth-death algorithm is described. Thus, we are now able to simulate from the density and we can therefore approximate statistics such as means and variances by sample means and variances,

$$\mathbb{E}_\psi g(X) = \frac{1}{m} \sum_{i=1}^{m} g(x_i),$$

where $x_1, \ldots, x_m$ are samples from the density with parameter ψ.

In the second case, when the parameter ψ in the density ratio are different we have

$$\frac{f_\psi(x)}{f_\theta(x)} = \frac{c(\theta)}{c(\psi)} \frac{h_\psi(x)}{h_\theta(x)} = \mathbb{E}_\psi \frac{h_\theta(X)}{h_\psi(X)} \frac{h_\psi(x)}{h_\theta(x)}, \tag{7}$$

where for ψ and θ close, the mean can be approximated by the sample mean over realizations from the model with parameter ψ, see e.g. Geyer (1999) In cases where the parameters are not sufficiently close, a bridge can be used, cf. Gelman and Meng (1998).

Ratios of the form (7) are used in Likelihood inference, where the Likelihood function is computed up to a fixed constant

$$\overline{L}(\psi; x) = \frac{L(\psi; x)}{L(\psi_0; x)} = \frac{f_\psi(x)}{f_{\psi_0}(x)}.$$

Notice that in the Strauss point process the parameters β and γ can be estimated by solving equations involving means, cf. (**??**), however, in order to estimate the parameter r, we need to compute the Likelihood function.

A.1. The Metropolis-Hastings birth-death algorithm

The Metropolis-Hastings birth-death algorithm is described in e.g. Møller (1999, Page 150). The algorithm is iterative and starts with the empty point configuration $x = \emptyset$. Then we update the configuration by adding a point, deleting a point, or do nothing. If the density f is a Markov density, c.f. e.g. **?** then the series of point patterns constitutes a Markov chain with distribution that converges towards f. Thus, running the chain infinitely, we end up with a point pattern with density f. However, in general the chain will reach equilibrium after a large but finite transitions (burn-in) and we can therefore stop the chain after, say `nburn`, transitions and then the point pattern can be considered to be a realization of the process with density f.

A simple, often used, version of the algorithm is to do as follows in each of the transitions. Let $x$be the current state of the Markov chain,

- With probability 0.5 we propose to add a point.
  A point ξ is chosen uniformly random in $\mathcal{X}$
  - With probability $\frac{f(x \cup \xi)}{f(x)} \frac{\text{Area}(\mathcal{X})}{n(x)+1}$ the state is changed to $x \cup \{\xi\}$
  - Else the state remains x
- Else we propose to delete a point.
  A point ξ is chosen uniformly random among the points in x
  - With probability $\frac{f(x \setminus \xi)}{f(x)} \frac{n(x)}{\text{Area}(\mathcal{X})}$ the state is changed to $x \setminus \{\xi\}$

– Else the state remains x

For the Strauss point process on the unit square we have,

$$\frac{f(x \cup \xi)}{f(x)} \frac{\text{Area}(\mathcal{X})}{n(x)+1} = \frac{\beta \gamma^{s(\xi;x)}}{n(x)-1} \quad \text{and} \quad \frac{f(x \backslash \xi)}{f(x)} \frac{n(x)}{\text{Area}(\mathcal{X})} = \frac{n(x)}{\beta \gamma^{s(\xi;x)}},$$

where

$$s(\xi; x) = \sum_{\eta \in x \backslash \xi} 1(\|\xi - \eta\| < r)$$

is the number of points in x which are closer than r units away from ξ.

In Figure 15 a program for a transition of above form is sketched. Thus, the lines "run x 1 step in MHBD algorithm" in the programs in Figures 2 and 3 can be replaced by `makeMHBDstep();`. Notice that lines 4–6 in Figure 3 constitutes the Metropolis-Hastings Birth-Death algorithm.

The function `makeMHBDstep` works as described above. The call `drand48()` returns a uniformly distributed real number in $(0; 1)$. Thus, with probability 0.5 we enter the first part of the `if...else` statement. The vector `data` is a vector with entries of the class `point` and the current point configuration occupies the places $0, \ldots,$`n`-1. Thus, the entry `n` is set to hold the proposed point and if we accept to add the point, then the number of points is just counted up by 1 (`n++` means `n=n+1`). The function `unifpoint()` is a member function of the class `point` and makes the current point uniform random in $\mathcal{X}$.

In the second half of the `if...else` statement we propose to delete one of the existing points (notice that if `n` is 0, then the acceptance probability is 0 and we keep the empty point configuration). The integer `etaIndex` is set to be uniformly among the integers $0, \ldots,$`n`-1. Thus, the point we propose to delete is the point from `data` with index `etaIndex`. If we accept to delete this point, it is replaced by last `data` point (index `n-1`) and `n` is counted 1 down.

The function `acceptProb` falls in two parts. The point to be added or deleted is contained at place `j` of `data`. If `j` equals `n` then we are in the process of adding a point, and otherwise the point in question is one of the points in the realization and we know that we are proposing to delete a point. In this way we can return the proper accept probabilities.

B. The *loess* algorithm

The *loess* algorithm is a method for predicting function values using local regression based on observed data.

```
void pointprocess :: makeMHBDstep ( void )
{
  if ( drand48() > 0.5 )
    {
      // PROPOSAL: ADD A UNIFORMLY CHOSEN POINT TO data

      data[n].unifpoint();

      if ( acceptProb( n ) > drand48() )
        n++;
    }
  else
    {
      // PROPOSAL: DELETE A UNIFORMLY CHOSEN POINT FROM data

      if ( n > 0 )
        {
          int etaIndex = (int) (drand48()*n);

          if ( acceptProb( etaIndex ) > drand48() )
            {
              data[etaIndex].replaceWith( data[n-1] );
              n--;
            }
        }
    }
}

double pointprocess :: acceptProb( int j )
{
  if ( j == n )                 // CASE: ADDING A POINT
   return  f(data∪data[n])/f(data) * Area(X)/(n+1);
  else                          // CASE: DELETING A POINT
   return  f(data\data[i])/f(data) * n/Area(X);
}
```

Figure 15: One step in a Metropolis-Hastings birth-death algorithm. The current point configuration is always the indexes $0 \ldots \mathtt{n}-1$ of the vector `data` with entries from the class `point`.

Let $\mathcal{X}$ be a ..... space and let $x_1, \ldots, x_n \in \mathcal{X}$. Suppose that for each x_i we have observed a value y_i. We suppose that there exist an underlying function f describing data such that

$$y_i = f(x_i) + \epsilon,$$

where ϵ is a stochastic error.

Given a point $x \in \mathcal{X}$, the *loess* algorithm is now a method for determining $f(x)$ based on the observed data (x_i, y_i), $i = 1, \ldots, n$.

Loess has two parameters $\alpha > 0$ and $\lambda \in \{1, 2\}$. We suppose that $\alpha < 1$. Then it determines the percentage of data points used in the prediction, thus, the αn data points with x_i closest to x will be used in the prediction with a weight $w_i(x)$ increasing in the distance from x.

The second parameter, λ, determines whether the prediction is based on linear or quadratic regression. Thus,

$$\hat{f}(x) = a + bx + cx^2 \quad \text{or} \quad \hat{f}(x) = a + bx.$$

The estimation of the regression coefficients are now based on the weighted least-squares, i.e. minimizing the variance

$$\sum_{i=1}^{n} w_i(x)(y_i - a - bx_i - cx_i^2)^2,$$

where $c = 0$ if $\lambda = 1$. See furthermore Cleveland (1993, Section 3.2) for very nice graphical as well as mathematical explanations.

Notice that α serves as a global smoothing parameter whereas λ is a local smoothing parameter.

At this time we use the implementation in the statistical software package `Splus`.

The predicted values of the means were computed in the same (β, γ) grid points using loess with parameters $\alpha = 0.1$ and $\lambda = 2$.

C. Using Beostar

Måske et afsnit med hvordan praktisk kører et parallelt program på Beostar: OBS

```
> lamboot
```

etc.

D. Scaling a Strauss point process

Let X be a Strauss point process on $\mathcal{X} = (0;1) \times (0;1)$ with parameters (β, γ, r), i.e. density of the form (1). Using the transformation result Jensen and Nielsen (2000, Proposition 3.2) with $h(\eta) = a\eta$, we get that $Y = h(X) = aX$ is a point process on $\mathcal{Y} = h(\mathcal{X}) = (0;a) \times (0;a)$ with density

$$\begin{aligned}
f_Y(y) &= f_X(h^{-1}(y))\mathrm{e}^{\mathrm{area}(\mathcal{Y})-\mathrm{area}(\mathcal{X})} \prod_{\eta \in y} Jh^{-1}(\eta) \\
&= c(\beta,\gamma,r)^{-1}\beta^{n(y/a)}\gamma^{s_r(y/a)}\mathrm{e}^{\mathrm{area}(\mathcal{Y})-\mathrm{area}(\mathcal{X})} \left(\frac{1}{a^2}\right)^{n(y)} \\
&= c(\beta,\gamma,r)^{-1}\mathrm{e}^{\mathrm{area}(\mathcal{Y})-\mathrm{area}(\mathcal{X})} \left(\frac{\beta}{a^2}\right)^{n(y)} \gamma^{s_{ar}(y)} \\
&= c\left(\frac{\beta}{a^2},\gamma,ar\right)^{-1} \left(\frac{\beta}{a^2}\right)^{n(y)} \gamma^{s_{ar}(y)},
\end{aligned}$$

where $y \in \Omega_{\mathcal{Y}}$ (the set of point patterns in $\mathcal{Y}$). Thus, $Y = aX$ is a Strauss process with parameters $\left(\frac{\beta}{a^2}, \gamma, ar\right)$. The last equality is shown using the extension formula for the Poisson point process, see Lieshout (2000, ...), and the coarea formula, see Jensen (1998, ...).

$$\begin{aligned}
c(\beta,\gamma,r) &= \int_{\Omega_{\mathcal{X}}} \beta^{n(x)}\gamma^{s_r(x)}\Pi(dx) \\
&= \sum_{n=0}^{\infty} \mathrm{e}^{-\mathrm{area}(\mathcal{X})} \frac{1}{n!} \int_{\mathcal{X}} \cdots \int_{\mathcal{X}} \beta^n \gamma^{s_r(\{x_1,\ldots,x_n\})} dx_1 \cdots dx_n \\
&= \sum_{n=0}^{\infty} \mathrm{e}^{-\mathrm{area}(\mathcal{X})} \frac{1}{n!} \int_{\mathcal{Y}} \cdots \int_{\mathcal{Y}} \beta^n \gamma^{s_r(\{h^{-1}(x_1),\ldots,h^{-1}(x_n)\})} \\
&\qquad\qquad Jh^{-1}(x_1)\cdots Jh^{-1}(x_n) dx_1 \cdots dx_n \\
&= \sum_{n=0}^{\infty} \mathrm{e}^{-\mathrm{area}(\mathcal{X})} \frac{1}{n!} \int_{\mathcal{Y}} \cdots \int_{\mathcal{Y}} \beta^n \gamma^{s_r(\{x_1/a,\ldots,x_n/a\})} \left(\frac{1}{a^2}\right)^n dx_1 \cdots dx_n \\
&= \mathrm{e}^{\mathrm{area}(\mathcal{Y})-\mathrm{area}(\mathcal{X})} \\
&\quad \sum_{n=0}^{\infty} \mathrm{e}^{-\mathrm{area}(\mathcal{Y})} \frac{1}{n!} \int_{\mathcal{Y}} \cdots \int_{\mathcal{Y}} \left(\frac{\beta}{a^2}\right)^n \gamma^{s_{ar}(\{x_1,\ldots,x_n\})} dx_1 \cdots dx_n \\
&= \mathrm{e}^{\mathrm{area}(\mathcal{Y})-\mathrm{area}(\mathcal{X})} \int_{\Omega_{\mathcal{Y}}} \left(\frac{\beta}{a^2}\right)^{n(x)} \gamma^{s_{ar}(x)} \Pi(dx) \\
&= \mathrm{e}^{\mathrm{area}(\mathcal{Y})-\mathrm{area}(\mathcal{X})} c\left(\frac{\beta}{a^2},\gamma,ar\right)
\end{aligned}$$

References

Baddeley, A. J. and Møller, J. (1989). Nearest-neighbour Markov point processes and random sets. *Int. Statist. Rev.*, 57:89–121.

Baddeley, A. J. and van Lieshout, M. N. M. (1995). Area-interaction point processes. *Ann. Inst. Statist. Math.*, 47:601–619.

Brix, A. and Møller, J. (1998). Space-time multitype log Gaussian Cox processes with a view to modelling weed data. *Research Report R–98–2012, Department of Mathematical Sciences, Aalborg University.* To appear in Scand. J. Statist., 2001.

Cleveland, W. S. (1993). *Visualizing data.* Hobart Press, Summit, New Jersey.

Diggle, P. J. (1983). *Statistical analysis of spatial point patterns.* Academic Press.

Gelman, A. and Meng, X.-L. (1998). Simulating normalizing constants: from importance sampling to bridge sampling to path sampling. *Statist. Sci.*, 13(2):163–185.

Geyer, C. J. (1999). Likelihood inference for spatial point processes. In Barndorff-Nielsen, O. E., Kendall, W. S., and van Lieshout, M. N. M., editors, *Stochastic Geometry: Likelihood and Computation*, chapter 3, pages 79–140. Chapman and Hall/CRC, London.

Hahn, U., Jensen, E. B. V., van Lieshout, M. N. M., and Nielsen, L. S. (2001). Inhomogeneous Markov point processes by location dependent scaling. *Research Report 16, Laboratory for Computational Stochastics, University of Aarhus.*

Jensen, E. B. V. (1998). *Local Stereology.* World Scientific, Singapore.

Jensen, E. B. V. and Nielsen, L. S. (2000). Inhomogeneous Markov point processes by transformation. *Bernoulli*, 6:761–782.

Jensen, E. B. V. and Nielsen, L. S. (2001). A review on inhomogeneous spatial point processes. In Basawa, I. V., Heyde, C. C., and Taylor, R. L., editors, *Selected Proceedings of the Symposium on Inference for Stochastic Processes*, volume 37 of *IMS Lecture Notes*, pages 297–318.

Lieshout, van, M. N. M. (2000). *Markov point processes and their applications.* World Scientific.

Møller, J. (1999). Markov chain Monte Carlo and spatial point processes. In Barndorff-Nielsen, O. E., Kendall, W. S., and van Lieshout, M. N. M., editors, *Stochastic Geometry: Likelihood and Computation*, chapter 4, pages 141–172. Chapman and Hall/CRC, London.

Nielsen, L. S. (2000). Modelling the position of cell profiles allowing for both inhomogeneity and interaction. *Image Analysis and Stereology*, 19(3):183–187.

Nielsen, L. S. (2001). *Point process models allowing for interaction and inhomogeneity*. PhD thesis, Dept. of Mathematical Sciences, University of Aarhus.

Ogata, Y. and Tanemura, M. (1986). Likelihood estimation of interaction potentials and external fields of inhomogeneous spatial point patterns. In Francis, I. S., Manly, B. F. J., and Lam, F. C., editors, *Proc. Pacific Statistical Congress – 1985*, pages 150–154. Amsterdam, Elsevier.

Pancake, C. M. (1996). Is parallelism for You? *IEEE Computational Science & Engineering*, 3(2):18–37.

Ripley, B. D. (1977). Modelling spatial patterns. *J. Roy. Statist. Soc. Ser. B*, 39(2):172–212. With discussion.

Ripley, B. D. and Kelly, F. P. (1977). Markov point processes. *J. London Math. Soc.*, 15:188–192.

Snir, M., Otto, S. W., Huss-Lederman, S., Walker, D. W., and Dongarra, J. (1996). *MPI: The complete reference*. The MIT Press.

Stoyan, D., Kendall, W. S., and Mecke, J. (1995). *Stochastic Geometry and its Statistical Applications*. Wiley, Chichester, second edition.

Stoyan, D. and Stoyan, H. (1998). Non-homogeneous Gibbs process models for forestry – a case study. *Biometrical Journal*, 40:521–531.

Strauss, D. J. (1975). A model for clustering. *Biometrika*, 62:467–475.