Day4 - Markov chain Monte Carlo methods

Michel Bje Randahl Nielsen (s093481)

July 1, 2015

1 About the implementations...

All implementations are done in F# scripts but uses functionality offered by the R programming environment. The scripts utilizes the type provider functionality which F# provides, to invoke functions in R.

F# project page: http://fsharp.org/

F# RProvider project page: https://bluemountaincapital.github.io/FSharpRProvider/

All scripts has been devloped inside visual studio with F# 3.1, but is only dependend on the existance of .Net 4.5 and a F# 3.1 instalation to be run. A full project has been attached to these hand ins and contains all of the scripts and nescesary dll's.

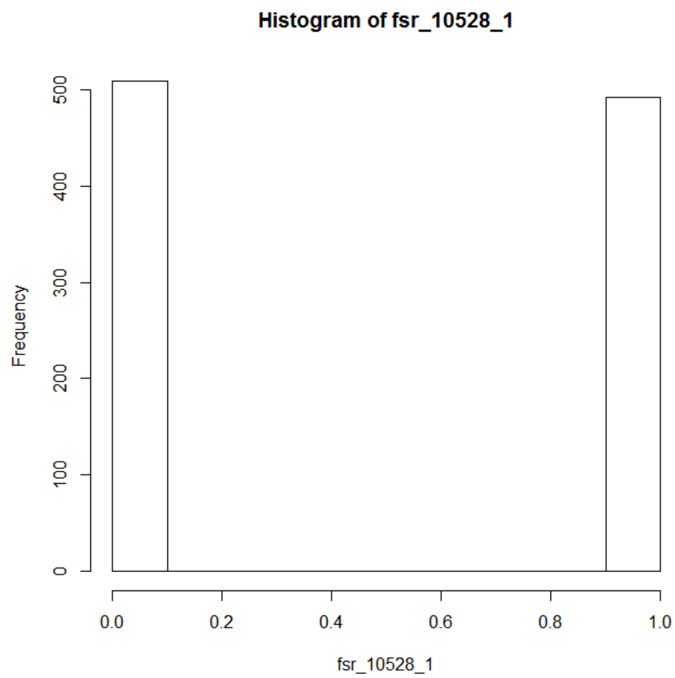
Note that all scripts must run following code in the beginning in order to function, but this piece of code has been omitted in these reports.

```
#r @"..\packages\R.NET.Community.1.5.16\lib\net40\RDotNet.dll"
#r @"..\packages\R.NET.Community.FSharp.0.1.9\lib\net40\RDotNet.FSharp.dll"
#r @"..\packages\RProvider.1.1.8\lib\net40\RProvider.dll"
#r @"..\packages\RProvider.1.1.8\lib\net40\RProvider.Runtime.dll"

open RDotNet
open RProvider
open RProvider.stats
open RProvider.graphics
open RProvider.mvtnorm
open RProvider.MASS
open System
```

2 Exercises

2.1 Simulate a Markov chain with values in S = {0,1} such that P(xi = 1—xi1 = 0) = 1/sqrt(2) and P(xi = 1—xi1 = 1) = 1/ Plot the result.

```
let markov_chain n =
    //defining the transition function for the markov chain
    let transition_fun = function
                //return 1 with a probability of 1/sqrt(2) if input is 0
        | 0 -> R.rbinom(1, 1, 1.0 / Math.Sqrt(2.0)).AsNumeric()
                |> Seq.head
                |> int
                //return 1 with a probability of 1.PI if input is 1
        | _ -> R.rbinom(1, 1, 1.0 / Math.PI).AsNumeric()
                |> Seq.head
                |> int

    //recursive loop to perform n-number of iterations of the markov chain
    let rec loop n' prev = seq {
        match n' with
                //end case
        | 0 -> yield transition_fun prev
                //loop case
        | _ -> let res = transition_fun prev
                yield res
                yield! loop (n'-1) res
    }

    //initializing the loop
    R.rbinom(1, 1, 0.5).AsNumeric()
    |> Seq.head
    |> int
    |> loop n

//running the markov chain and plotting the result in a histogram
markov_chain 1000
|> R.hist
```



The probability of ending up in a state of 0 is 0.51 and the probability of ending up in a state of 1 is 0.49.

2.2 Simulate a Markov chain with the Metropolis algorithm with the Cauchy distribution C(0,1) as a target distribution. Use e.g. a normal pdf with 0 mean and a unit variance as a proposal q.

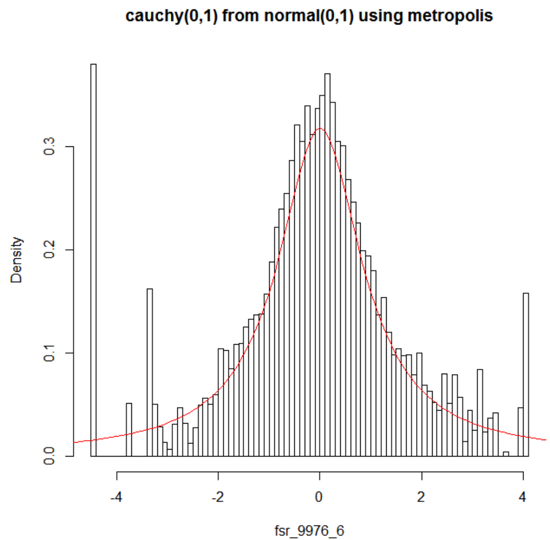
Defining the markov chain function, transition function, auxillery funciton based on normal distribution, target function using the cauchy distribution and the itteration loop. The metropolis hastings algorithm works by rejecting or accepting a sequence of random values from an auxillery distribution (in this case normal dist). The rejection is done as a basis of the previous accepted value which is used in the calculaton of the metropolis ratio that determines the probability for which a proposed value should be accepted or not.

```
let markov_chain n =
    //target function
    let f (x: float) =
        R.dcauchy(x, 0, 1).AsNumeric()
        |> Seq.head

    //auxillery function
    let q (x: float) =
        R.dnorm(x, mean=0, sd=1).AsNumeric()
        |> Seq.head

    //transition function using the basic metropolis algorithm
    let transition_fun prev =
        //candidate value
        let y = R.rnorm(1, mean=0, sd=1).AsNumeric()
                |> Seq.head

        //compute metropolis ratio
        let met_ratio = f y  * q prev / (f prev * q y)

        let accept_prob = Seq.min [1.0; met_ratio]
```

```
        //toss unfair coin with accept prob
        let accept = R.rbinom(1, 1, accept_prob).AsNumeric()
                     |> Seq.head
                     |> int
        match accept with
        | 0 -> prev
        | _ -> y

    //recurive loop to perform n' itterations
    let rec loop n' prev = seq {
        match n' with
               //end case
        | 0 -> yield transition_fun prev
               //loop case
        | _ -> let res = transition_fun prev
               yield res
               yield! loop (n' - 1) res
    }

    //initializing the loop
    R.rnorm(1, mean=0, sd=1).AsNumeric()
    |> Seq.head
    |> loop n

let n = 10000
let result = markov_chain n |> List.ofSeq
```

Plotting the data in a histogram.

```
namedParams [
    "x", box result
    "breaks", box 100
    "main", box "cauchy(0,1) from normal(0,1) using metropolis"
    "probability", box true
]
|> R.hist
let xs = [-5.0 .. 0.1 .. 5.0]
let normal_fun =
    xs
    |> List.map (fun x -> Seq.head(R.dcauchy(x, 0, 1).AsNumeric()))
namedParams [
    "x", box xs
    "y",box normal_fun
    "col", box "red"
]
|> R.lines
```



It can be seen that the resulting distribution follows the cauchy dist in the center, but fails to follow the dist in the extremes.

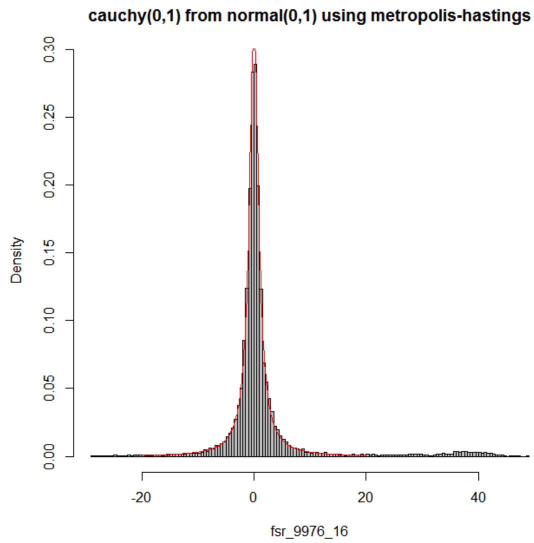
2.3 Same as previous exercise with a Metropolis-Hastings algorithm with Gaussian proposal. How does the simulated Markov chain depend on the variance of the proposal distribution?

```
//the difference between metropolis-hastings and the basic metropolis
   algorithms are that the metropolis-hastings algorithm calculates the
   auxillery function based on the proposed state and the previous state,
   and that the proposed value is drawn from a distribution with a mean
   based on the previous accepted value
let markov_chain n =
    //target function
    let f (x: float) =
        R.dcauchy(x, 0, 1).AsNumeric()
        |> Seq.head

    //auxillery function
    let q (x: float) (y: float) =
        R.dnorm(x, mean=y, sd=1).AsNumeric()
        |> Seq.head

    //transition function using the metropolis-hastings algorithm
    let transition_fun prev =
        //candidate value
        let y = R.rnorm(1, mean=prev, sd=1).AsNumeric()
                |> Seq.head

        //compute metropolis ratio
        let met_ratio = f y  * q prev y / (f prev * q y prev)

        let accept_prob = Seq.min [1.0; met_ratio]
        //toss unfair coin with accept prob
        let accept = R.rbinom(1, 1, accept_prob).AsNumeric()
```

```
                     |> Seq.head
                     |> int
        match accept with
        | 0 -> prev
        | _ -> y

    //recursive loop to perform n iterations in the markov chain
    let rec loop n' prev = seq {
        match n' with
               //end case
        | 0 -> yield transition_fun prev
               //loop case
        | _ -> let res = transition_fun prev
               yield res
               yield! loop (n' - 1) res
    }

    R.rnorm(1, mean=0, sd=1).AsNumeric()
    |> Seq.head
    |> loop n

let n = 25000
let result = markov_chain n |> List.ofSeq
```

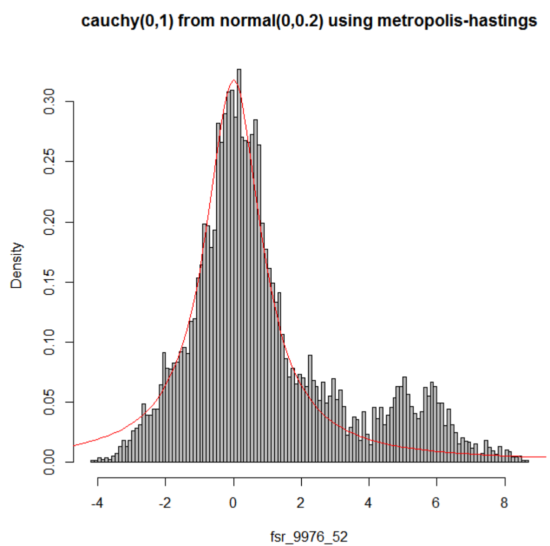
Plotting the data in a histogram.

```
namedParams [
    "x", box result
    "breaks", box 150
    "main", box "cauchy(0,1) from normal(0,1) using metropolis-hastings"
    "col", box "grey"
    "probability", box true
]
|> R.hist
let xs = [-20.0 .. 0.1 .. 20.0]
let normal_fun =
    xs
    |> List.map (fun x -> Seq.head(R.dcauchy(x, 0, 1).AsNumeric()))
namedParams [
    "x", box xs
    "y",box normal_fun
    "col", box "red"
]
|> R.lines
```



The histogram reveals that the sampled data has reached its target distribution very well.

To get a sense of the effect that the variance has on the result, we generate a sample using an auxillery function based on the normal dist but with a very low variance.



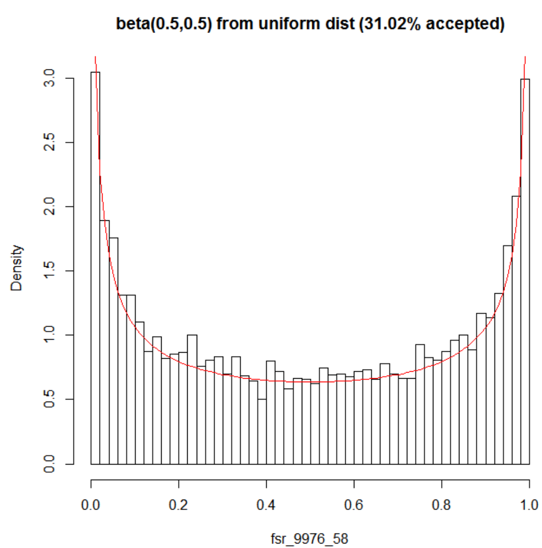
Here it can be seen that a very low variance will result in a skewed distribution that doesn't follow the target distribution.

2.4 Simulate a sample from a Beta B(1/2,1/2) with the rejection algorithm (using e.g. the uniform distribution as majorating density). Then simulate the same B(1/2,1/2) with the MH algorithm.

First simulating using the rejection method.

```
//simulating using the rejection algorithm
let n = 25000
let U1 = R.runif(n).AsNumeric()
let U2 = R.runif(n).AsNumeric()

let accept_function scaling_factor (u1, u2) =
```

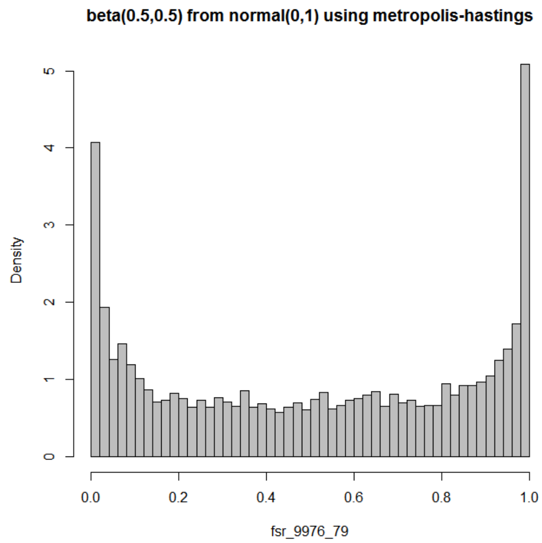
```
    let beta_res =
        R.dbeta(u1, shape1=0.5, shape2=0.5).AsNumeric()
        |> Seq.head
    scaling_factor * u2 < beta_res

let rbeta_rej =
    Seq.zip U1 U2
    |> Seq.filter (accept_function 3.0)
    |> Seq.map fst
let accepted = 100.0 * float(rbeta_rej |> Seq.length) / float n

//plotting the result of the rejection algorithm
namedParams [
    "x", box rbeta_rej
    "main", box (sprintf "beta(0.5,0.5) from uniform dist (%A%% accepted)"
       accepted)
    "breaks", box 50
    "probability", box true
]
|> R.hist
let xs = [0.0 .. 0.01 .. 1.0]
namedParams [
    "x", box xs
    "y",box <| R.dbeta(xs, 0.5, 0.5)
    "col", box "red"
]
|> R.lines
```



Next simulating using the MH algorithm.

```
//simulating using the MH algorithm
let markov_chain n =
    //target funciton
    let f (x: float) =
        R.dbeta(x, 0.5, 0.5).AsNumeric()
```

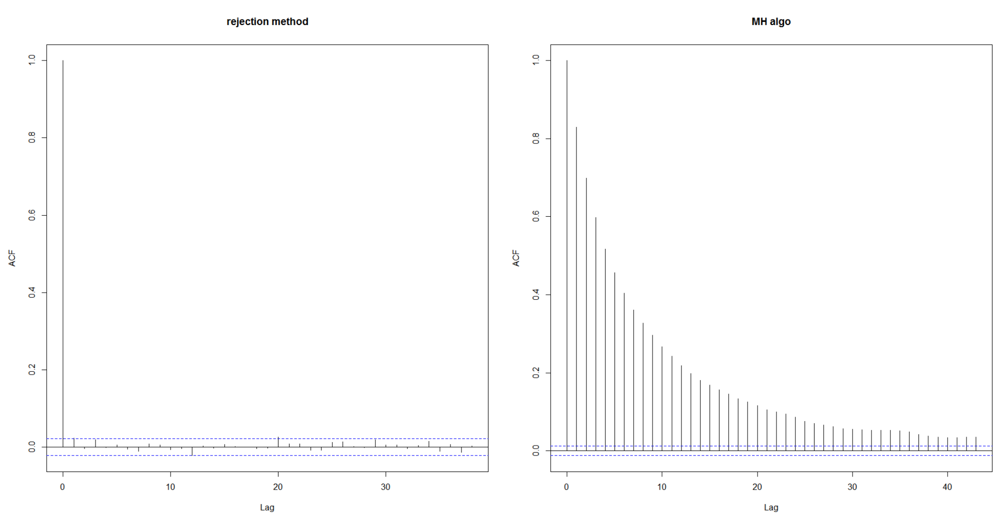
```
        |> Seq.head

    //auxillery function
    let q (x: float) (y: float) =
        R.dnorm(x, mean=y, sd=1).AsNumeric()
        |> Seq.head

    //transition function based on MH algorithm
    let transition_fun prev =
        let y = R.rnorm(1, mean=prev, sd=1).AsNumeric()
                |> Seq.head
        let met_ratio =
            f y * q prev y /
            (f prev * q y prev)
        let accept_prob = Seq.min [1.0; met_ratio]
        let accept = R.rbinom(1, 1, accept_prob).AsNumeric()
                     |> Seq.head
                     |> int
        match accept with
        | 0 -> prev
        | _ -> y
    let rec loop n' prev = seq {
        match n' with
        | 0 -> yield transition_fun prev
        | _ ->
            let res = transition_fun prev
            yield res
            yield! loop (n' - 1) res
    }

    R.rnorm(1, mean=0, sd=1).AsNumeric()
    |> Seq.head
    |> loop n

//calculating and plotting the result
let rbeta_mh = markov_chain n |> List.ofSeq
namedParams [
    "x", box <| Seq.filter (fun x -> x >= 0.0) rbeta_mh
    "breaks", box 50
    "main", box "beta(0.5,0.5) from normal(0,1) using metropolis-hastings"
    "col", box "grey"
    "probability", box true
]
|> R.hist
namedParams [
    "x", box xs
    "y",box <| R.dbeta(xs, 0.5, 0.5)
    "col", box "red"
]
|> R.lines
```



based on the histograms alone, it is difficult to spot any difference between the two results.

- Plotting the autocorrelation.

```
//plotting the autocorrelation
namedParams [
    "mfrow",[1;2]
]
|> R.par

namedParams [
    "x", box rbeta_rej
    "main", box "rejection method"
]
|> R.acf
namedParams [
    "x", box rbeta_mh
    "main", box "MH algo"
]
|> R.acf
```

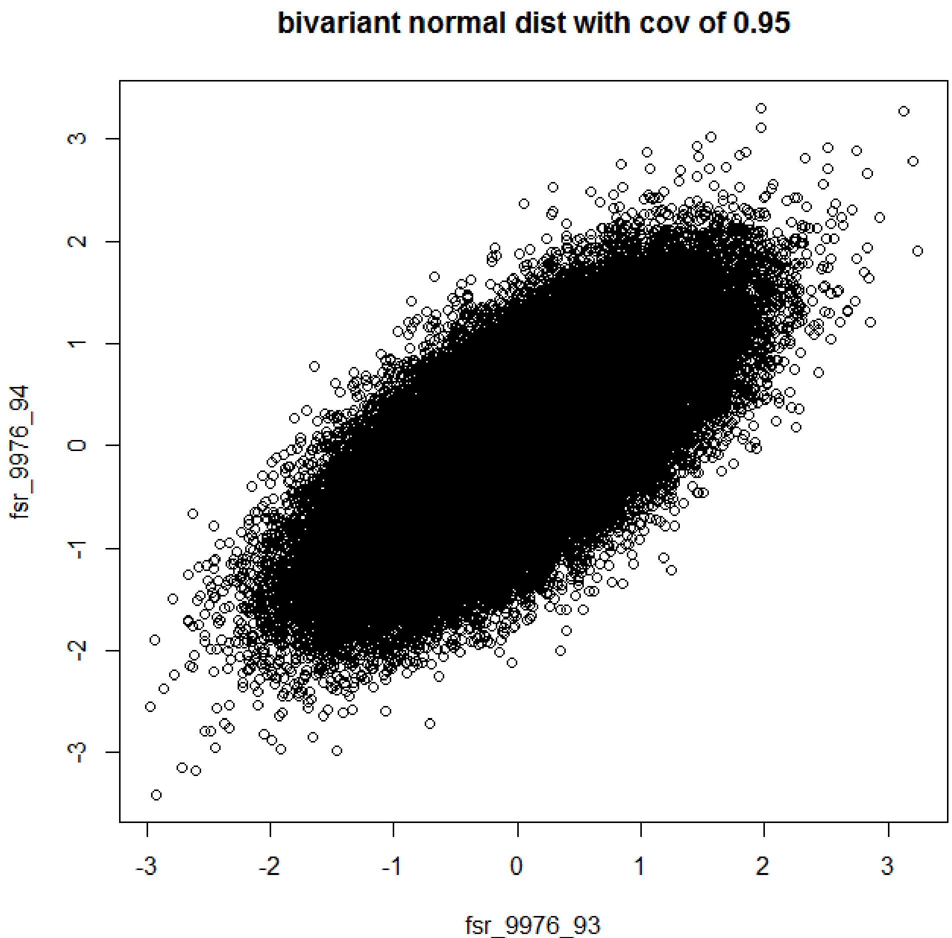


from the autocorrelation plots, it is obvious that the MH algorithm produces data that is highly autocorrelated *and the rejection algorithm does not produce autocorrelated data. The autocorrelation seen in the MH generated data, is clearly due to the nature of markov processes where the state is dependend on the previous state.*

2.5 Simulate a sample of size 1000 using the Gibbs sampler

simulating a MVN sample with mean 0 and covariance matrix with rho=0.7 using the gibbs sampler.

```
let n = 100000
//choosing a rho of 0.7
let rho = 0.7

let gibbs_sampler n rho =
    //gibbs sampling function
    let gibbs_sample x' =
        let y = R.rnorm(1, rho*x', 1.0 - rho**2.0 ).AsNumeric()
                |> Seq.head
                //generate x using y to adjust the mean
        let x = R.rnorm(1, rho*y, 1.0 - rho**2.0 ).AsNumeric()
                |> Seq.head
        x,y

    //recursive loop for the gibbs sampling process
    let rec loop n' x' = seq {
        match n' with
               //end case
        | 0 -> yield gibbs_sample x'
               //loop case
        | _ -> let res = gibbs_sample x'
               yield res
               yield! loop (n'-1) (fst res)
    }

    //initialize loop
    R.rnorm(1, 0, 1).AsNumeric()
    |> Seq.head
    |> loop n

let result = gibbs_sampler n rho
             |> List.ofSeq
             |> List.unzip
result
||> fun X Y ->
    namedParams [
        "x", box X
        "y", box Y
        "main", box "bivariant normal dist with cov of 0.95"
    ]
|> R.plot
```



Testing that the covariance matrix of the sample data is similiar to the target covariance.

```
//testing the covariance matrix
//if the relationships in the covariance matrix matches the relationships
   from the original covariance matrix, then we have succeeded the
   simulation
let cov =
    result
    ||> fun X Y ->
        array2D [X ; Y]
    |> R.t
    |> fun x -> R.cov(x).AsNumericMatrix().ToArray()

//calculate the rho value
let rho' = (cov.[0,1] + cov.[1,0]) / (cov.[0,0] + cov.[1,1])
```

Outputs...

```
val cov : float [,] = [[0.5070249569; 0.3543349309]
                       [0.3543349309; 0.5078870188]]
val rho' : float = 0.6982574636
```

The estimated rho' value is similiar to the target rho value and the proportions of the covariance matrix is similiar to the target covariance matrix.