Day5 - Monte Carlo integration methods & Monte Carlo optimization 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.

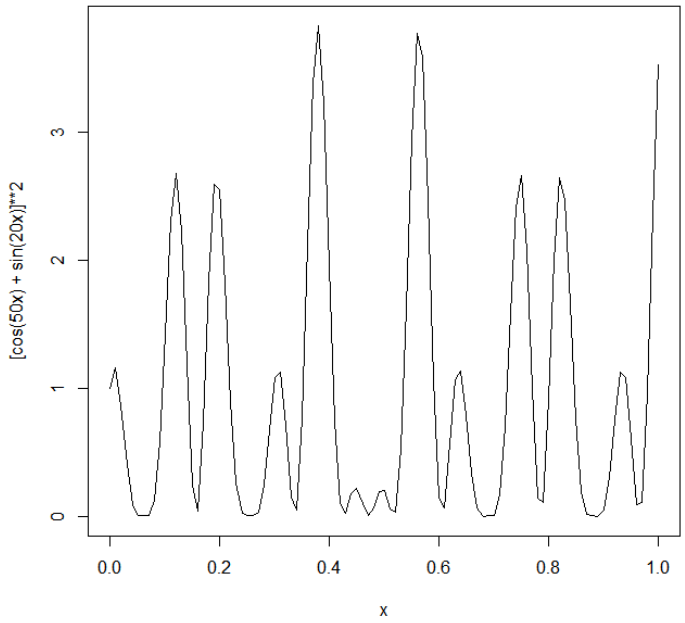
```
#r @"..\packages\R.NET.Community.1.5.16\lib\net40\RDotNet.dll"
#r @"..\packages\R.NET.Community.FSharp.0.1.9\lib\net40\RDotNet.FSharp.dll"
#r @"..\packages\RProvider.1.1.8\lib\net40\RProvider.dll"
#r @"..\packages\RProvider.1.1.8\lib\net40\RProvider.Runtime.dll"

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

2 Exercises on Monte Carlo integration I

2.1 Consider the function h(x) = $[cos(50x) + sin(20x)]^2$

- Plot the function on $[0; 1]$**.**

```
let f x = (Math.Cos(50.0 * x) + Math.Sin(20.0 * x))**2.0

//Plot this function on [0,1]
let xs = [0.0 .. 0.01 .. 1.0]
namedParams [
    "x", box xs
    "y", box <| Seq.map f xs
    "type", box "l"
    "xlab", box "x"
    "ylab", box "[cos(50x) + sin(20x)]**2"
]
|> R.plot
```



- write algorithm to compute the integral and compare with the R function integrate(). *I choose to calculate the integral, by performing a uniform random sample over the function in the range* $[0; 1]$ *and then compute the mean.*

```
let n = 10000
R.runif(n).AsNumeric()
|> Seq.map f
|> Seq.average
```

Outputs...

```
val it : float = 0.9680233396
```

using the R function integral().

```
//Compare your result to those obtained with the R function integrate()
R.eval(R.parse(text="hx <- function(x) {(cos(x*50) + sin(20*x))**2}"))
//R.integrate(f="hx", lower=0, upper=1).AsList().Names
R.integrate(f="hx", lower=0, upper=1).AsList().["value"].AsNumeric()
|> Seq.head
```

Outputs...

```
val it : float = 0.9652009361
```

the results of the two integrals are very close.

2.2 Computing the volume of the unit ball

Defining the function to calculate the volume of a unit ball. The algorithm basically samples a set of random vectors, calculates the ratio of the vectors inside and outside the ball, and using this ratio to calculate the volume as $2^d * ratio$ *.*

```
let n = 10000

//function to calcualte exact volume of a unit ball
let unit_ball_true_volume dim =
    Math.PI**(float(dim)/2.0) /
        (R.gamma(1.0 + float(dim)/2.0).AsNumeric()
        |> Seq.head)

//function to estimate volume of a unit ball
let unit_ball_volume_estimate simulations dimensions =
    let identity_fun xs =
        xs
        |> Seq.map (fun x -> x**2.0)
        |> Seq.sum
        |> fun x ->
            match x <= 1.0 with
            | true -> 1.0
            | false -> 0.0

    let U = [|for i in 1 .. dimensions ->
              R.runif(simulations, -1, 1).AsNumeric().ToArray()|]
            |> array2D

    let scored_vectors =
        [0 .. simulations-1]
        |> Seq.map (fun n -> U.[0..,n], identity_fun U.[0..,n])

    let points_in_ball =
        scored_vectors
        |> Seq.filter (fun x -> snd x > 0.0)
        |> Seq.map fst
        |> array2D

    let point_count =
        points_in_ball
        |> Array2D.length1
```

```
    [for n in 0 .. (point_count-1) ->
     points_in_ball.[n,0], points_in_ball.[n,1]]
    |> List.ofSeq
    |> List.unzip
    ||> fun X Y ->
        namedParams [
            "x", box X
            "y", box Y
            "xlim", box [-1;1]
            "ylim", box [-1;1]
        ]
    |> R.plot
    |> ignore

    let ball_ratio = scored_vectors |> Seq.map snd |> Seq.average
    let estimate = 2.0**float(dimensions) * ball_ratio
    let variance =
        1.0 / float(simulations)**2.0 *
            ([for x in scored_vectors ->
              if snd x > 0.0 then 1.0 else 0.0]
             |> Seq.map (fun x -> (x - estimate)**2.0)
             |> Seq.sum)
    estimate, variance
```

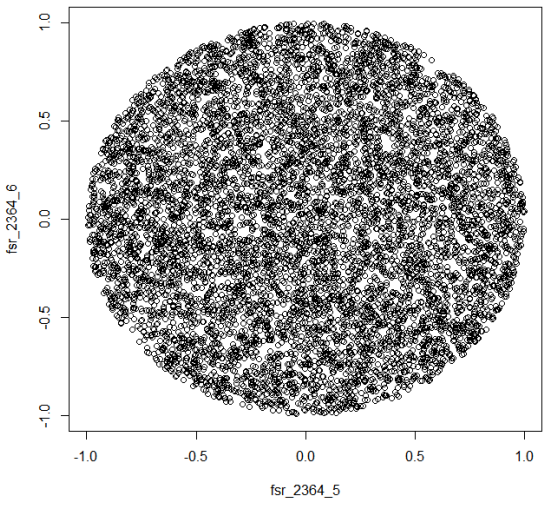
we can now test this function on the dimensions 2,3 and 10.
For two dimensions..

```
unit_ball_volume_estimate n 2
```

Outputs and plots..

```
val it : float * float = (3.1336, 0.000569312448)
```

where the first value is the estimated volume and the second is the variance.



we compare this to the volume calculated by the more exact function.

```
unit_ball_true_volume 2
```

Outputs...

```
val it : float = 3.141592654
```

For three dimensions..

```
unit_ball_volume_estimate n 3
unit_ball_true_volume 3
```

Outputs...

```
val it : float * float = (4.2224, 0.001389929632)
val it : float = 4.188790205
```

For ten dimensions..

```
unit_ball_volume_estimate n 10
unit_ball_true_volume 10
```

Outputs...

```
val it : float * float = (1.6384, 0.000268071168)
val it : float = 2.55016404
```

The results for dimensions 2 and 3 are very similiar indicating that the algorithm makes good estimations, however for 10 dimensions the algorithm seems to under estimate a lot. This can be accounted for simply by making the random sample size bigger, however that will also require a lot more computation.

2.3 Modify the previous method by simulating a symmetric multivariate normal

```
//function to estimate volume of unit ball by simulating from a symmetrix
   multivariate normal dist
let unit_ball_volume_estimate2 (simulations: int) (dimensions: int) =
    let sigma = R.diag(dimensions).AsNumericMatrix().ToArray()

    let identity_fun xs =
        xs
        |> Seq.map (fun x -> x**2.0)
        |> Seq.sum
        |> fun x ->
            match x <= 1.0 with
            | true ->
                let dmvnorm = R.dmvnorm(xs).AsNumeric()
                              |> Seq.head
                1.0 / dmvnorm
            | false -> 0.0

    let mv_random_normal_dist =
        namedParams [
            "n", box (dimensions * simulations)
            "mean", box <| R.rep(0, dimensions)
        ]
        |> R.rmvnorm
```

```
    let U =
        namedParams [
            "data", box mv_random_normal_dist
            "ncol", box dimensions
        ]
        |> fun x -> R.t(R.matrix(x)).AsNumericMatrix().ToArray()

    let scored_vectors =
        [0 .. simulations-1]
        |> Seq.map (fun n -> U.[0..,n], identity_fun U.[0..,n])

    let points_in_ball =
        scored_vectors
        |> Seq.filter (fun x -> snd x > 0.0)
        |> Seq.map fst
        |> array2D
    let point_count =
        points_in_ball
        |> Array2D.length1

    [for n in 0 .. (point_count-1) ->
     points_in_ball.[n,0], points_in_ball.[n,1]]
    |> List.ofSeq
    |> List.unzip
    ||> fun X Y ->
        namedParams [
            "x", box X
            "y", box Y
            "xlim", box [-1;1]
            "ylim", box [-1;1]
        ]
    |> R.plot
    |> ignore

    let estimate = scored_vectors |> Seq.map snd |> Seq.average
    let variance =
        1.0 / float(simulations)**2.0 *
            ([for x in scored_vectors -> snd x]
             |> Seq.map (fun x -> (x - estimate)**2.0)
             |> Seq.sum)
    estimate, variance
```

testing the function on the 10 dimension unit ball.

```
//testing the function on the unit balls 2,3,10
unit_ball_volume_estimate2 n 2
unit_ball_volume_estimate2 n 3
unit_ball_volume_estimate2 n 10
```

Outputs...

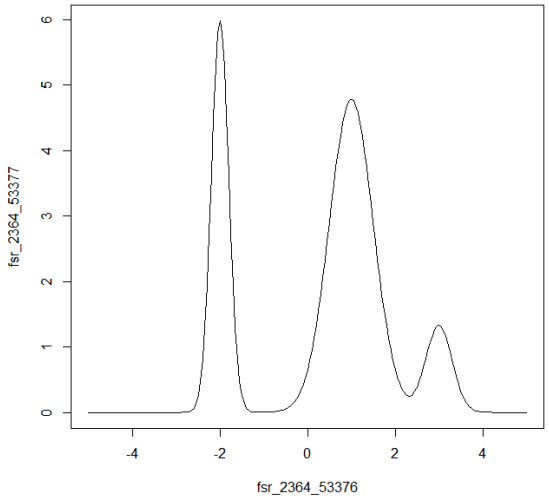
```
val it : float * float = (3.122353568, 0.001568811627)
val it : float * float = (4.194401536, 0.007246264817)
val it : float * float = (6.053649184, 9.178616339)
```

Again the algorithm overshoots the unit ball with 10 dimensions, on the other dimensions the difference is minscule.

3 optimization methods - Exercises I

3.1 Consider h dened in R as h ¡- function(x){3*dnorm(x,m=-2,sd=.2) + 6*dnorm(x,m=1,sd=.5) + 1*dnorm(x,m=3,sd=.3)}

```
let fx (x: float) =
    3.0 * (R.dnorm(x, mean = -2, sd = 0.2).AsNumeric() |> Seq.head) +
    6.0 * (R.dnorm(x, mean = 1, sd = 0.5).AsNumeric() |> Seq.head) +
    (R.dnorm(x, mean = 3, sd = 0.3).AsNumeric() |> Seq.head)
let xs = [-5.0 .. 0.05 .. 5.0]
namedParams [
    "x", box xs
    "y", box <| Seq.map fx xs
    "type", box "l"
]
|> R.plot
```



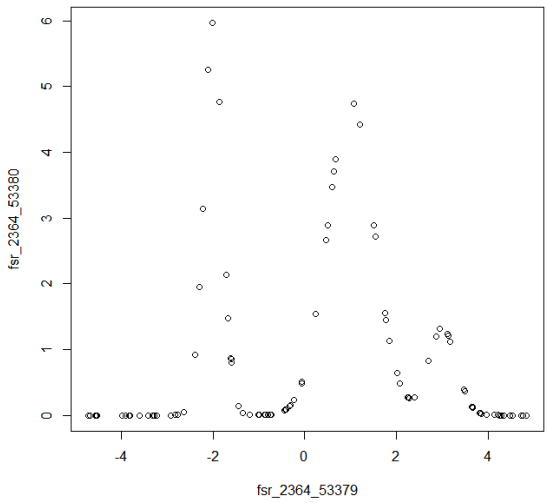
- Find the maximum of h on [5,5] using the uniform simulation method and a sample of size 100.

```
//sampling 100 random uniformly dist values and running them through the
   given function
let uniform_sample =
    R.runif(100, min = -5, max = 5).AsNumeric()
    |> List.ofSeq
    |> List.map (fun x -> x, fx x)

let maxu = uniform_sample |> List.maxBy snd

uniform_sample
|> List.unzip
||> fun X Y ->
    namedParams [
        "x", box X
        "y", box Y
    ]
|> R.plot
```

Outputs...

```
val maxu : float * float = (-2.01551616, 5.966152743)
```

where the first value is the argument to the function and the second is the corresponding output of the function.



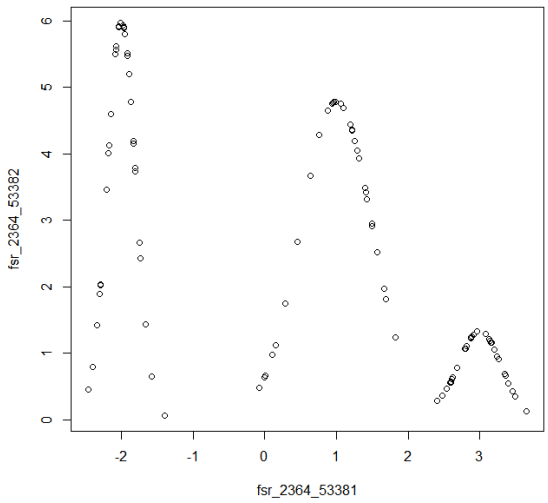
- Find the maximum of h on [5,5] using a non-uniform simulation method.

```
//the mixture distribution is created by sampling from three normal dists
let mixture_dist_sample =
    [R.rnorm(33, mean = -2, sd = 0.2).AsNumeric()
     R.rnorm(33, mean = 1, sd = 0.5).AsNumeric()
     R.rnorm(33, mean = 3, sd = 0.3).AsNumeric()]
    |> Seq.concat
    |> Seq.map (fun x -> x, fx x)

let maxm = mixture_dist_sample |> Seq.maxBy snd

mixture_dist_sample
|> List.ofSeq
|> List.unzip
||> fun X Y ->
    namedParams [
        "x", box X
        "y", box Y
    ]
|> R.plot
```

Outputs...

```
val maxm : float * float = (-1.972755701, 5.928869398)
```

where the first value is the argument to the function and the second is the corresponding output of the function.



- Compare your result to the output of the R function optimize.

```
//[1.3] - Compare your result to the output of the R function optimize
let r_optimize =
    R.eval(R.parse(text="optimise(function(x) {3*dnorm(x,m = -2,sd = 0.2) +
       6*dnorm(x,m = 1,sd = 0.5) + 1*dnorm(x,m = 3,sd = 0.3)}, lower = -5,
       upper = 5, maximum = TRUE)"))
     .AsList()
     .["objective"]
     .AsNumeric()
    |> Seq.head
```

Outputs...

```
val r_optimize : float = 4.787307365
```

The built in funciton in R clearly under estimates and gets stuck in a local maxima. The random simulation algorithms seems to perform better for the given function.

3.2 Implement the simulated annealing method

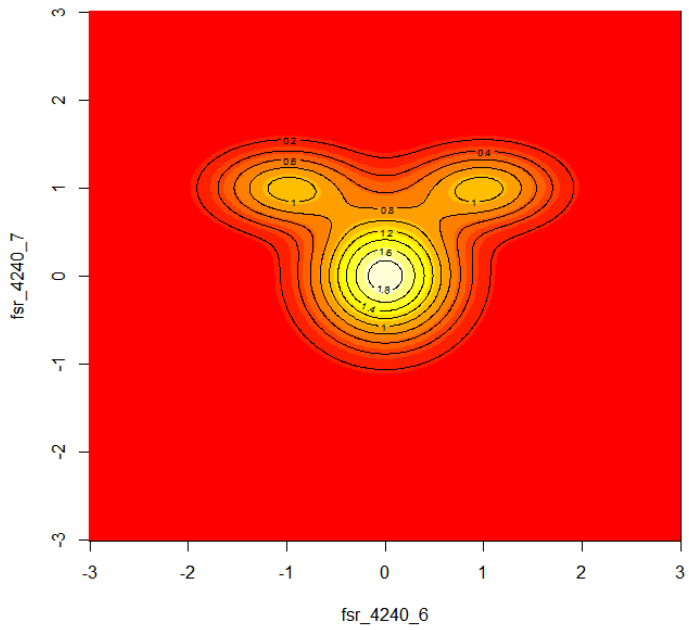
defining the funciton to analyze in R and F#

```
let r_fun = """
            h = function(x,y)
            {
                3* dnorm(x,m=0,sd=.5)*dnorm(y,m=0,sd=.5) +
                dnorm(x,m=-1,sd=.5)*dnorm(y,m=1,sd=.3) +
                dnorm(x,m=1,sd=.5)*dnorm(y,m=1,sd=.3)
            }
            """
R.eval(R.parse(text = r_fun))

let h (x: float, y: float) =
    3.0 * (R.dnorm(x, mean =  0, sd = 0.5).AsNumeric() |> Seq.head) *
          (R.dnorm(y, mean =  0, sd = 0.5).AsNumeric() |> Seq.head) +

    1.0 * (R.dnorm(x, mean = -1, sd = 0.5).AsNumeric() |> Seq.head) *
```

```
          (R.dnorm(y, mean =  1, sd = 0.3).AsNumeric() |> Seq.head) +

    1.0 * (R.dnorm(x, mean =  1, sd = 0.5).AsNumeric() |> Seq.head) *
          (R.dnorm(y, mean =  1, sd = 0.3).AsNumeric() |> Seq.head)
```

plotting the function

```
let xs = [-3.0 .. 0.01 .. 3.0]
let ys = [-3.0 .. 0.01 .. 3.0]
let H = R.outer(xs, ys, FUN = "h").AsNumericMatrix().ToArray()
namedParams [
    "x", box xs
    "y", box ys
    "z", box H
]
|> R.image
namedParams [
    "x", box xs
    "y", box ys
    "z", box H
    "add", box true
]
|> R.contour
```



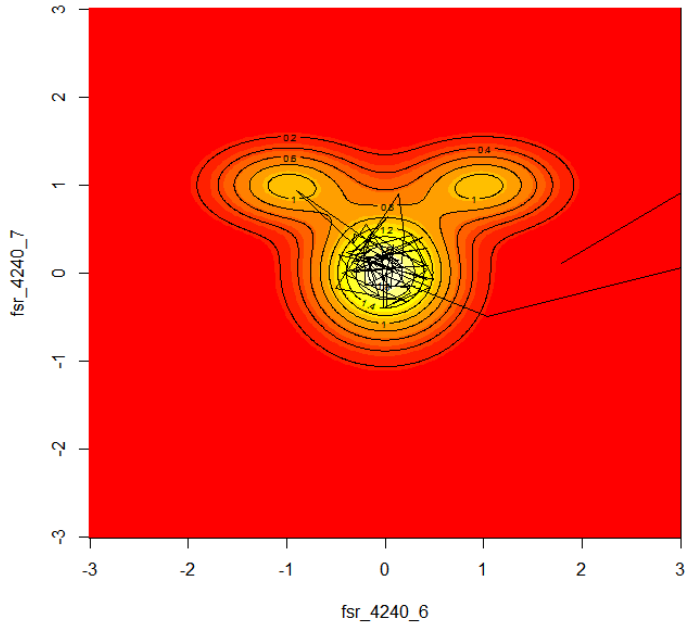
Defining the simulated annealing algorithm to be used to find the maximum of the function

```
//simulated annealing algorithm
let n = 1000

let sim_ann n h =
    let rec loop (n': int) (prev: float * float) = seq {
        if n' < n then
            //random vector
```

```
            let zeta = R.rnorm(2,0,1).AsNumeric() |> List.ofSeq
            let x,y = zeta.[0], zeta.[1]

            //limiting values to interval [-3;3]
            let limit v =
                match v with
                | v when v <= -3.0 -> -3.0
                | v when v >= 3.0  -> 3.0
                | _ -> v

            //adding random vector to the previous step
            let candidate =
                limit(fst prev + zeta.[0]),
                limit(snd prev + zeta.[1])

            let delta_h = (h candidate) - (h prev)
            let Temperature = 1.0 / Math.Log(float n')
            let accept_prob = Seq.min [1.0; Math.Exp(delta_h / Temperature)]

            //accepting based on unfair coin with accept_prob
            let accept = R.rbinom(1, 1, accept_prob).AsNumeric() |> Seq.head

            if accept = 1.0 then
                yield candidate
                yield! loop (n' + 1) candidate
            else
                yield! loop (n' + 1) prev
        }

    let initial = R.runif(2, -3, 3).AsNumeric() |> List.ofSeq
    loop 1 (initial.[0], initial.[1])
```

Invoking the algorithm and plotting the result

```
let result = sim_ann n h |> List.ofSeq
let result_arr =
    result
    |> Seq.map (fun x -> [fst x; snd x])
    |> array2D
namedParams [
    "x", box result_arr
    "type", box "l"
]
|> R.lines
```



getting the last point and the best point produced by the algorithm

```
let last_point_amplitude =
    (List.ofSeq result).[(Seq.length result) - 2]
    h (List.ofSeq result).[(Seq.length result) - 2]
let best_point =
    result
    |> Seq.map (fun x -> x, h x)
    |> Seq.maxBy snd
```

Outputs...

```
val last_point_amplitude : float = 1.084278069
val best_point : (float * float) * float =
  ((0.01406406395, 0.03323956101), 1.906488315)
```

we can see that the algorithm finds a very good estimate for the global maxima within the given range, but ends up in a local maxima for the last point.