David Šiška School of Mathematics Sciences, University of Edinburgh

**2018/19 Semester 1**

# Object Oriented Programming with Applications

## Problem Sheet 3 - Wednesday 17th October 2018[1]

**Exercise 3.1.** You might have noticed that the following code for computing the Fibonacci sequence fails (or takes very long to run) for $n > 100$.

```
static ulong NumberWithoutHashtable(ulong n)
{
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return NumberWithoutHashtable(n - 1) + NumberWithoutHashtable(n - 2);
}
```

Use either a `Hashtable` or another data structure to store already computed values so that you can calculate the Fibonacci numbers even for large $n$.

[1] Last updated 25th October 2018

*Solution.* We will keep track of the computed numbers in the following way.

```
using System;
using System.Collections;

namespace HashFibonacci
{

    class Fibonacci
    {
        static private Hashtable alreadyComputed;

        static Fibonacci()
        {
            alreadyComputed = new Hashtable ();
        }

        static ulong NumberWithoutHashtable(ulong n)
        {
            if (n == 0)
                return 0;
            else if (n == 1)
                return 1;
            else
                return NumberWithoutHashtable(n - 1) + NumberWithoutHashtable(n - 2);
        }

        static public ulong Number(ulong n)
        {
            if (n == 0)
                return 0;
            else if (n == 1)
                return 1;
            else {
                // try to look it up in the Hashtable
                if (alreadyComputed.ContainsKey (n))
                    return (ulong)alreadyComputed [n];

                // compute it recursively, store it
                else {
                    ulong fibNum = Number (n - 1) + Number (n - 2);
                    alreadyComputed.Add (n, fibNum);
                    return fibNum;
                }
            }
        }
    }


    class MainClass
    {
        public static void Main (string[] args)
        {
            ulong n = 200;
            ulong fib = Fibonacci.Number (n);
            Console.WriteLine ("Fibonnaci number {0} is {1}.", n, fib);
        }
    }
}
```

**Exercise 3.2.** The linear least squares problem consists of finding

$$\hat{\beta} = \operatorname*{arg\,min}_{\beta \in \mathbb{R}^n} |y - X\beta|^2,$$

where $X$ is a given $m \times n$ matrix such that $X_{i1} = 1$, $i = 1, \ldots, m$ and $y$ is a column vector in $\mathbb{R}^m$. It can be shown (using calculus) that the solution is given by solving the linear system

$$(X^T X)\hat{\beta} = X^T y.$$

Use `MathNet.Numerics` linear algebra methods to complete the class below which is suggested for solving the problem

```
using System;
using MathNet.Numerics.LinearAlgebra;

class LinearLeastSquares
{
    private Matrix<double> X;
    private Vector<double> y;

    public LinearLeastSquares(Matrix<double> dataX, Vector<double> dataY)
    {
        y = dataY;
        int n = dataX.ColumnCount + 1;
        int m = dataX.RowCount;
        X = Matrix<double>.Build.Dense (m, n);
        X.SetColumn(0, Vector<double>.Build.Dense(m,1.0));
        for (int j = 1; j < n; ++j)
            X.SetColumn (j, dataX.Column (j-1));
    }

    public Vector<double> CalculateCoefficients()
    {
```

Here you need to write code to solve $(X^T X)\hat{\beta} = X^T y$ and return $\hat{\beta}$.

```
    }
}
```

Now test it by adding the following.

```
class MainClass
{
    public static void Main (string[] args)
    {
        double[,] x = {{ 1.0} , {2.0}, {3.0}, {4.0} };
        Matrix<double> dataX = Matrix<double>.Build.DenseOfArray (x);

        double[] y = {6, 5, 7, 10};
        Vector<double> dataY = Vector<double>.Build.DenseOfArray (y);

        LinearLeastSquares lls = new LinearLeastSquares (dataX, dataY);
        Vector<double> beta = lls.CalculateCoefficients ();
        Console.WriteLine (beta);
    }
}
```

*Solution.* The missing lines are

```
Matrix<double> M = X.Transpose() * X;
Vector<double> rhs = X.Transpose() * y;
return M.Solve (rhs);
```

**Exercise 3.3.** Use the lecture slides on `System.Numerics` and Excel integration with ExcelDNA to create two new Excel functions:

1. `public static double ComplexLogarithmRealPart(double realPart, double imaginaryPart)` which returns, for $z \in \mathbb{C}$, $\Re(\ln(z))$ i.e. the real part of complex logarithm,

2. `public static double ComplexLogarithmImaginaryPart(double realPart, double imaginaryPart)` which returns, for $z \in \mathbb{C}$, $\Im(\ln(z))$ i.e. the imaginary part of complex logarithm.

For a real number $z = \exp(i\theta)$ use these functions to plot the real and imaginary parts of the complex logarithm for $\theta \in [0, 2\pi]$.

*Solution.* You will need to add the reference to the `System.Numerics` library as shown on lecture slides. You will also need to start with ExcelDNA template from the course website. Once you have those, the following methods do the job.

```csharp
using ExcelDna.Integration;
using System.Numerics;

namespace TestExcelDNA
{
    public static class MyFunctions
    {
        [ExcelFunction(Description = "Complex Logarithm Real Part")]
        public static double ComplexLogarithmRealPart(double realPart, double imaginaryPart)
        {
            Complex z = new Complex(realPart, imaginaryPart);
            return Complex.Log(z).Real;
        }

        [ExcelFunction(Description = "Complex Logarithm Imaginary Part")]
        public static double ComplexLogarithmImaginaryPart(double realPart, double imaginaryPart)
        {
            Complex z = new Complex(realPart, imaginaryPart);
            return Complex.Log(z).Imaginary;
        }
    }
}
```