David Šiška School of Mathematics Sciences, University of Edinburgh

2018/19 Semester 2

Object Oriented Programming with Applications

Problem Sheet 4 - Wednesday 24th October 2018[1]

Exercises marked with an asterisk (*) need to be handed in by noon on 23rd October

Exercise 4.1. Get a working implementation of the `CompositeIntegrator` class. Use the lecture material.
You will notice that the class, as presented in lectures is not *robust*. Consider and modify your code according for the following situations (i.e. throw appropriate exceptions).

1. What if `newtonCotesOrder` in the is a number other than $1, 2, 3, 4$?
2. What if `N` in the `Integrate` method is less than or equal to 0?
3. Does the code work if `a` $>$ `b` in the `Integrate` method?

Solution. For first part we throw an exception in the constructor. For second part we throw and exception in the integrate method. For the third part we note that the code behaves correctly since if $a > b$ then

$$\int_b^a f(x)dx = -\int_a^b f(x)dx.$$

```
public class CompositeIntegrator
{
    private int newtonCotesOrder;
    const int maxOrder = 4;
    const int maxOrderLength = 5;
    private int numOfEvals;

    private static double[,] weights = new double[maxOrder, maxOrderLength] {
        {0.5, 0.5, 0, 0, 0},  // Trapezium rule
        {1.0/6.0, 4.0/6.0, 1.0/6.0, 0, 0},// Simpson's rule
        {1.0/8.0, 3.0/8.0, 3.0/8.0, 1.0/8.0,0},
        {7.0/90.0, 32.0/90.0, 12.0/90.0, 32.0/90.0, 7.0/90.0}
    };
    private double[] quadraturePoints;
    private double[] quadraturePointsFVal;

    public CompositeIntegrator()
    {
        newtonCotesOrder = 1;
        quadraturePoints = new double[newtonCotesOrder+1];
        quadraturePointsFVal = new double[newtonCotesOrder+1];
    }

    public CompositeIntegrator(CompositeIntegrator integrator)
    {
        newtonCotesOrder = integrator.newtonCotesOrder;
        quadraturePoints = new double[newtonCotesOrder+1];
        quadraturePointsFVal = new double[newtonCotesOrder+1];
    }

    public CompositeIntegrator(int newtonCotesOrder)
```

[1] Last updated 13th November 2018

```
    {
        if (newtonCotesOrder <= 0 || newtonCotesOrder > maxOrder)
            throw new System.ArgumentException ("Only order 1,2,3,4 are supported");
        this.newtonCotesOrder = newtonCotesOrder;
        quadraturePoints = new double[newtonCotesOrder+1];
        quadraturePointsFVal = new double[newtonCotesOrder+1];
    }

    private void updateQuadraturePointsAndFvals(double a, double h,Func<double, double> f)
    {
        double delta = h / newtonCotesOrder;
        for (int i = 0; i <= newtonCotesOrder; i++) {
            quadraturePoints [i] = a + i * delta;
            quadraturePointsFVal[i] = f(quadraturePoints[i]);
            numOfEvals++;
        }
    }


    public double Integrate(Func<double, double> f, double a, double b, int N)
    {
        if (N <= 0)
            throw new System.ArgumentException ("Number of partitions must be an integer > 0.");
        double integral = 0;
        double h = (b - a) / N;
        for (int i = 0; i < N; i++)
        {
            updateQuadraturePointsAndFvals (a+i*h, h, f);
            double stepIncrement = 0.0;
            for (int j = 0; j <= newtonCotesOrder; j++) {
                stepIncrement += weights[newtonCotesOrder-1,j] * quadraturePointsFVal [j];
            }
            integral += stepIncrement*h;
        }
        return integral;
    }

    public int GetNumEvals() { return numOfEvals; }
    public void ResetNumEvals() { numOfEvals = 0; }
}
```

Exercise 4.2. * The Newton's method for approximating solutions x to $f(x) = 0$, where $f : \mathbb{R} \to \mathbb{R}$ is assumed to be differentiable is an iterative method where, given an initial guess for the solution x_0 the next entry in the sequence of approximations is calculated as:

$$x_n := x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}\,, \quad n = 1, 2, \ldots\,.$$

Do the following:

1. To see that you understand how this works calculate the first three approximations to $0 = x^2 - 2 =: f(x)$ with $x_0 = 2$ (this is a nice way of approximating $\sqrt{2}$ if you ever have to do this by hand).
2. Create a method with the following "signature":

   ```
   static double NewtonSolver(Func<double, double> f,
                              Func<double, double> fPrime,
                              double x0,
                              double maxError, int maxIter)
   ```

   It should have the following properties: the return value should be the approximation x_n of solution to $f(x) = 0$ such that $|f(x)| <$`maxError`. If the number of iterations has reached or exceeded `maxIter` then an exception should be thrown. If `fPrime != null` then it should be used, otherwise the derivative should be approximated using symmetric finite differences. That is, for a given $\delta > 0$ we say that

   $$f'(x_0) \quad \text{is approximately} \quad \frac{1}{2\delta}\left(f(x_0 + \delta) - f(x_0 - \delta)\right).$$

3. Test it by approximating the solution to $0 = x^2 - 2$ with $x_0 = 2$.

Solution: Notice that if the solver fails to converge to a good approximation of a solution after the maximum number of iterations then it throws an exception, rather than returning a possible incorrect result.

```
static double NewtonSolver(Func<double, double> f,
        Func<double, double> fPrime,
        double x0, double maxError, int maxIter)
{
    double delta = maxError * 10;
    if (fPrime == null)
    {
        fPrime = (x) => {
            return (f (x + delta) - f (x - delta)) / (2 * delta);
        };
    }
    double x1 = x0;
    for (int iter = 0; iter < maxIter; iter++)
    {
        double fPrimeX0;
        fPrimeX0 = fPrime (x0);
        x1 = x0 - f (x0) / fPrimeX0;
        double error = Math.Abs (f (x1));
        if (error <= maxError)
            return x1;
        x0 = x1;
    }
    throw new System.AggregateException ("Newton solver failed to converge.");
}
```

Exercise 4.3. * Newton's method generalises to higher dimensions. Indeed consider a differentiable $F : \mathbb{R}^d \to \mathbb{R}^d$. Let $J_F(x)$ denote the Jacobian matrix of this function evaluated at x. That is:

$$J_F(x) = \begin{pmatrix} \frac{\partial F_1}{\partial x_1} & \frac{\partial F_1}{\partial x_2} & \dots & \frac{\partial F_1}{\partial x_d} \\ \frac{\partial F_2}{\partial x_1} & \frac{\partial F_2}{\partial x_2} & \dots & \frac{\partial F_2}{\partial x_d} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F_d}{\partial x_1} & \frac{\partial F_d}{\partial x_2} & \dots & \frac{\partial F_d}{\partial x_d} \end{pmatrix}.$$

Given an initial guess $x_0 \in \mathbb{R}^d$ we obtain successive approximations to x such that $F(x) = 0$ by solving

$$J_F(x_{n-1})(x_n - x_{n-1}) = -F(x_{n-1}), \;\; n = 1, 2, \dots .$$

Your task is to complete the class below:

```
public class NewtonSolver
{
    private const double delta = 1e-8; // for approximating partial derivatives

    private double tol;
    private int maxIt;

    public NewtonSolver(double tolerance, int maximumIterations)
    {
        tol = tolerance;
        maxIt = maximumIterations;
    }
    public Matrix<double> ApproximateJacobian(Func<Vector<double>,
                                        Vector<double>> F, Vector<double> x)
    {
    /* ... write the code ...*/
    }

    public Vector<double> Solve(Func<Vector<double>, Vector<double>> F,
                  Func<Vector<double>, Matrix<double>> J_F, Vector<double> x_0)
```

```
    {
    /* ... write the code ...*/
    }
}
```

The Newton Method should stop if either `maxIt` is reached or if the l^2 norm (i.e. the usual Euclidean norm) of $F(x_n)$ is smaller than `tol`.
Now test it by approximating the solution to $F(x, y) = 0$ where

$$F(x, y) := \begin{pmatrix} x^2 + y^2 - 2xy - 1 \\ x^2 - y^2 - 7 \end{pmatrix}$$

with $x_0 = (1, -1)^T$. Note that it is easy to see that at least one solution is $x = 4, y = 3$.
Hints.

- To define something like $F : \mathbb{R}^d \to \mathbb{R}^d$ in C# use:

```
Func<Vector<double>, Vector<double>> F = (x) =>
{
    Vector<double> y = Vector<double>.Build.Dense(x.Count);
    y[0] = x[0]*x[0] + x[1]*x[1] - 2*x[0]*x[1] - 1;
    y[1] = x[0]*x[0] - x[1]*x[1] - 7;
    return y;
};
```

- To define something like the Jacobian Matrix use:

```
Func<Vector<double>,Matrix<double>> J_F = (x) => {
   int d = x.Count;
   Matrix<double> J_F_vals = Matrix<double>.Build.Dense (d, d);
   J_F_vals[0,0] = 2*x[0] -2*x[1];
   J_F_vals[0,1] = 2*x[1]-2*x[0];
   J_F_vals[1,0] = 2*x[0];
   J_F_vals[1,1] = -2*x[1];
   return J_F_vals;
};
```

- You can see above how to define an empty vector and matrix respectively.
- To fill in a whole jth column in a matrix you can use: `M.SetColumn(j,v)` assuming `M` is a matrix and `v` is a vector of appropriate length.
- To solve a linear system $Ax = b$ (with A a $d \times d$ matrix and b a vector in $\mathbb{R}^d$, x unknown) use: `Vector<double> x = A.Solve(b);`.
- To find a determinant of a matrix A use `double determinant = A.Determinat();` Note that `A.Solve(b)` might also fail when the determinant is very small rather than exactly 0.

Solution. The code is on the course website. Please note that the solver throws an exception if maximum number of iterations is reached but the approximation is not within tolerance.

Exercise 4.4. In fact our interest in the Newton's method is also for minimization problems. A differentiable function $f : \mathbb{R}^d \to \mathbb{R}$ will have a local minimum (or maximum or saddle point) at x if

$$\nabla f(x) := \begin{pmatrix} \frac{\partial f}{\partial x_1}(x) \\ \vdots \\ \frac{\partial f}{\partial x_d}(x) \end{pmatrix}$$

is equal to zero. If f is convex then this will be the global minimum.

So we are approximating solutions to $F(x) = \nabla f(x) = 0$ using Newton's method.
The Jacobian matrix of F is now

$$\begin{pmatrix} \frac{\partial F_1}{\partial x_1} & \frac{\partial F_1}{\partial x_2} & \cdots & \frac{\partial F_1}{\partial x_d} \\ \frac{\partial F_2}{\partial x_1} & \frac{\partial F_2}{\partial x_2} & \cdots & \frac{\partial F_2}{\partial x_d} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F_d}{\partial x_1} & \frac{\partial F_d}{\partial x_2} & \cdots & \frac{\partial F_d}{\partial x_d} \end{pmatrix} = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_d} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_d} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_d \partial x_1} & \frac{\partial^2 f}{\partial x_d \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_d^2} \end{pmatrix}$$

which is the Hessian matrix of f.
In some cases one would have to approximate the Hessian using finite differences. Let us define $T_{\delta,i} y : \mathbb{R}^d \to \mathbb{R}^d$ as

$$T_{\delta,i} y = (y_1, \ldots, y_{i-1}, y_i + \delta, y_{i+1}, \ldots, y_d)^T.$$

Then

$$\begin{aligned} &\frac{\partial^2 f}{\partial x_i \partial x_j}(x) \\ &\approx \frac{1}{2h}\left(\frac{\partial f}{\partial x_j}(T_{i,h}x) - \frac{\partial f}{\partial x_j}(T_{i,-h}x)\right) \\ &\approx \frac{1}{2h}\left(\frac{1}{2h}\left(f(T_{j,h}T_{i,h}x) - f(T_{j,-h}T_{i,h}x)\right) - \frac{1}{2h}\left(f(T_{j,h}T_{i,-h}x) - f(T_{j,-h}T_{i,-h}x)\right)\right). \end{aligned}$$

Your task is to develop a Newton based minimizer by completing the class below

```
public class NewtonMnimizer
{
    private const double delta = 1e-7; // for approximating grad
    private NewtonSolver solver;

    public NewtonMnimizer(double tol, int maxIt)
    {
        solver= new NewtonSolver(tol, maxIt);
    }
    private Vector<double> ApproximateGrad(Func<Vector<double>, double> f,
                                           Vector<double> x)
    {
    /* ... write the code ... */
    }
    private Matrix<double> ApproximateHessian(Func<Vector<double>, double> f,
                                              Vector<double>> grad_f,
                                              Vector<double> x)
    {
    /* ... write the code, allow for grad_f == null ... */
    }
    public Vector<double> Minimize(Func<Vector<double>, double> f,
                            Func<Vector<double>, Vector<double>> grad_f,
                            Func<Vector<double>, Matrix<double>> hessian_f,
                            Vector<double> x_0)
    {
    /* ... write the code,
    allow for grad_f == null and also hessian_f == null ... */
    }
}
```

Test it by finding the global minimum of the function f given by $f(x, y) = x^2 + y^2$ by running the following code:

```
NewtonMnimizer minimizer = new NewtonMnimizer (1e-5, 100);
Func<Vector<double>, double> f = (x) => x [0] * x [0] + x [1] * x [1];

Vector<double> startPt = Vector<double>.Build.Dense (2);
```

```
startPt[0] = 1; startPt[1] = -1;

Console.WriteLine ("With approximate grad of f and hessian of f using f.d.");
Console.WriteLine(minimizer.Minimize(f, null, null, startPt));

Console.WriteLine ("With exact grad_f and approximate hessian f using f.d.");
Func<Vector<double>, Vector<double>> grad_f = (x) => {
    Vector<double> grad = Vector<double>.Build.Dense(x.Count);
    grad[0] = 2*x[0]; grad[1] = 2*x[1];
    return grad;
};
Console.WriteLine(minimizer.Minimize(f, grad_f, null, startPt));

Console.WriteLine ("With exact grad of f and hessian of f.");
Func<Vector<double>, Matrix<double>> hessian_f = (x) => {
    Matrix<double> hessian = Matrix<double>.Build.Dense(x.Count,x.Count);
    hessian[0,0] = 2; hessian[0,1] = 0;
    hessian[1,0] = 0; hessian[1,1] = 2;
    return hessian;
};
Console.WriteLine(minimizer.Minimize(f, grad_f, hessian_f, startPt));
Console.ReadKey ();
```

Solution. The code is on the course website. Please note that the solver throws an exception if maximum number of iterations is reached but the approximation is not within tolerance.