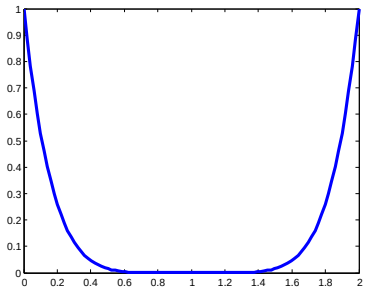
Solutions to Part B of Problem Sheet 2

Solution (2.4) In order to apply Newton's method, we need to know the derivatives of f. These are given by

$$f'(x) = 6(x-1)^5, \quad f''(x) = 30(x-1)^4.$$

We implement the function $f(x)$ and its derivatives in Python as follows

In [1]:
```
def f(x):
    return (x-1)**6

def df(x):
    return 6*(x-1)**5

def ddf(x):
    return 30*(x-1)**4
```



Newton's method is then the iteration

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} = x_k - \frac{1}{5}(x_k - 1) = 0.8x_k + 0.2.$$

To run Newton's method by hand, we choose a starting point x_0 and then compute x_1, x_2, etc. For example, if we start with $x_0 = 2$, we get the iterates $x_1 = 1.8, x_2 = 1.64$, and so on. Eventually the iterates will approach 1, but the process is slow.

We then run Newton's method, implemented as follows, with the three stopping criteria and with tolerance $\varepsilon = 10^{-6}$.

In [2]:
```
def newton(f, df, ddf, x0, tol, maxiter=100):
    """
    Newton's method
    """
    x, xold = x0, x0+2*tol*np.ones(x0.shape)
    iter = 0
    while ( la.norm(x-xold)>tol ) and ( iter < maxiter ):
        grad = df(x)
        hess = ddf(x)
        z = la.solve(hess,grad)[0]
        xold = x
        x = x-z
        iter += 1
    return x, iter
```

The results are:

(a) Iterations: 67, Solution found: $\overline{x} = 1.0000$;

(b) Iterations: 25, Solution found: $\overline{x} = 1.0425$;

(c) Iterations: 100, Solution found: $\overline{x} = 1.0000$.

While (b) was the fastest, the accuracy was not so great. Stopping criterium (a) is reasonable, since in view of the quadratic convergence, the inequality

$$\|\boldsymbol{x}_{k+1} - \boldsymbol{x}_k\| \geq \|\boldsymbol{x}_k - \boldsymbol{x}^*\| - \|\boldsymbol{x}_{k+1} - \boldsymbol{x}^*\| \geq \|\boldsymbol{x}_k - \boldsymbol{x}^*\| - M\|\boldsymbol{x}_k - \boldsymbol{x}^*\|^2$$

shows that the difference of successive iterates gives a good bound on the error. Criterium (b) is problematic, as the function can have a very flat slope while still being far away from the minimizer, as the example shows.

This issue is related to the notion of conditioning of the function f. Criterium (c) always has the same running time, and 100 iterations would normally be more than enough for Newton's method. However, this is a very pessimistic bound, and takes much longer than necessary.

Solution (2.5) The code is recreated below. We first define the methods, then the function, after that the parameters and run the code. In the end, the results are plotted. The code uses the first and second derivatives of the function f, which are rather complicated. They are given as

$$\nabla f(\boldsymbol{x}) = \begin{pmatrix} e^{x_1+3x_2-0.1} + e^{x_1-3x_2-0.1} - e^{-x_1-0.1} \\ 3e^{x_1+3x_2-0.1} - 3e^{x_1-3x_2-0.1} \end{pmatrix},$$

and

$$\nabla^2 f(\boldsymbol{x}) = \begin{pmatrix} e^{x_1+3x_2-0.1} + e^{x_1-3x_2-0.1} + e^{-x_1-0.1} & 3e^{x_1+3x_2-0.1} - 3e^{x_1-3x_2-0.1} \\ 3e^{x_1+3x_2-0.1} - 3e^{x_1-3x_2-0.1} & 9e^{x_1+3x_2-0.1} + 9e^{x_1-3x_2-0.1} \end{pmatrix}.$$

In [3]:

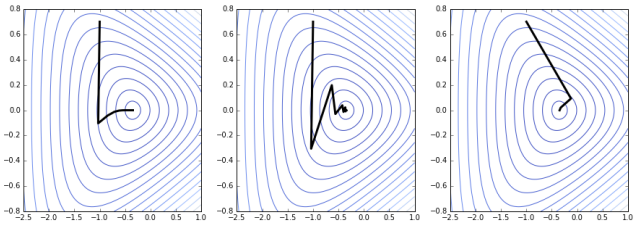
```
def newton(f, df, ddf, x0, tol, maxiter=100):
    """
    Newton's method with stopping criteria (a)
    """
    x = np.vstack((x0+2*tol*np.ones(x0.shape),x0)).transpose()
    i = 1
    while ( la.norm(x[:,i]-x[:,i-1]) > tol ) and ( i < maxiter ):
        grad = df(x[:,i])
        hess = ddf(x[:,i])
        z = la.solve(hess,grad)
        xnew = x[:,i]-z
        x = np.concatenate((x,xnew.reshape((len(x0), 1))), axis=1)
        i += 1
    return x[:,1:]

def graddesc_bt(f, df, ddf, x0, tol, maxiter=100, rho=0.5, c=0.1):
    """
    Gradient descent with backtracking
    """
    x = np.vstack((x0+2*tol*np.ones(x0.shape),x0)).transpose()
    i = 1
    while ( la.norm(x[:,i]-x[:,i-1]) > tol ) and ( i < maxiter ):
        p = -df(x[:,i])
        # Start backtracking
        alpha = 1
        xnew = x[:,i] + alpha*p
        while (f(xnew) >= f(x[:,i]) + alpha*c*np.dot(p, df(x[:,i]))):
            alpha = alpha*rho
            xnew = x[:,i] + alpha*p
        x = np.concatenate((x,xnew.reshape((len(x0),1))), axis=1)
        i += 1
    return x[:,1:]

def graddesc_co(f, df, ddf, x0, tol, maxiter=100):
    """
    Gradient descent with constant step length
    """
    alpha = 0.1
    x = np.vstack((x0+2*tol*np.ones(x0.shape),x0)).transpose()
    i = 1
    while ( la.norm(x[:,i]-x[:,i-1]) > tol ) and ( i < maxiter ):
        r = df(x[:,i])
        xnew = x[:,i] - alpha*r
        x = np.concatenate((x,xnew.reshape((len(x0),1))), axis=1)
        i += 1
    return x[:,1:]
```

In [4]:

```
def f(x):
    return np.exp(x[0]+3*x[1]-0.1)+np.exp(x[0]-3*x[1]-0.1)+np.exp(-x[0]-0.1)

def df(x):
    return np.array([np.exp(x[0]+3*x[1]-0.1)+np.exp(x[0]-3*x[1]-0.1)-np.exp(-x[0]-0.1),
                     3*np.exp(x[0]+3*x[1]-0.1)-3*np.exp(x[0]-3*x[1]-0.1)])

def ddf(x):
    return np.array([[np.exp(x[0]+3*x[1]-0.1)+np.exp(x[0]-3*x[1]-0.1)+np.exp(-x[0]-0.1),
                      3*np.exp(x[0]+3*x[1]-0.1)-3*np.exp(x[0]-3*x[1]-0.1)],
                     [3*np.exp(x[0]+3*x[1]-0.1)-3*np.exp(x[0]-3*x[1]-0.1),
                     9*np.exp(x[0]+3*x[1]-0.1)+9*np.exp(x[0]-3*x[1]-0.1)]])
```

In [5]:
```
tol = 1e-6
x0 = np.array([-1.,0.7])

# Run the three methods
xbt = graddesc_bt(f, df, ddf, x0, tol)
xco = graddesc_co(f, df, ddf, x0, tol)
xn = newton(f, df, ddf, x0, tol)
```

In [6]:
```
xvals = np.array([[np.linspace(-4,-0.5,20)], [np.zeros(20)]])
yvals = list(reversed(f(xvals)[0]))

# Create a meshgrid and a contour plot
xx = np.linspace(-2.5,1,100)
yy = np.linspace(-0.8,0.8,100)
X, Y = np.meshgrid(xx, yy)
# The construction inside looks odd: we want to transform the set of input pairs given
# by the meshgrid into a 2 x n array of values that we can apply f to (calling f on such
# an array will apply the function f to each column)
Z = f(np.dstack((X,Y)).reshape((X.size, 2)).transpose())
# the result of applying f is a long list, but we want a matrix
Z = Z.reshape(X.shape)

% matplotlib inline
cmap = plt.cm.get_cmap("coolwarm")

fig, ax = plt.subplots(1, 3, figsize=(15, 5))

ax[0].contour(X, Y, Z, yvals, cmap = cmap)
ax[0].plot(xco[0,:], xco[1,:], color='black', linewidth=3)

ax[1].contour(X, Y, Z, yvals, cmap = cmap)
ax[1].plot(xbt[0,:], xbt[1,:], color='black', linewidth=3)

ax[2].contour(X, Y, Z, yvals, cmap = cmap)
ax[2].plot(xn[0,:], xn[1,:], color='black', linewidth=3)

plt.show()
```



The number of iterations with constant step length is 44, the number of iterations with backtracking is 28, while the number of iterations with Newton's method is 6.

Solution (2.6)

(a) The verification follows from applying the definition of convexity.

(b) The code below minimizes the function using Newton's method with starting point $(a, b) = (0.5, 0.5)$ and tolerance 10^{-8}. The number of iterations is 6 and

the solution is $(a, b) = (1.724, -4.434)$. Plotting the probability of passing the exam as a function of the number of preparation hours, with the parameters (a, b), gives the curve in Figure 1.

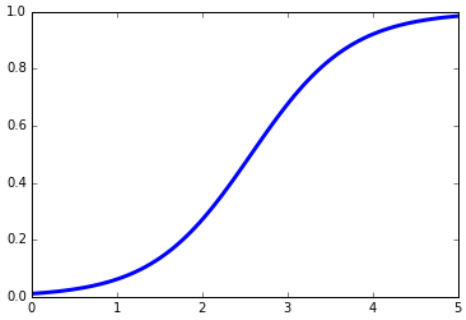


Figure 1: Probability of passing exam as function of preparation