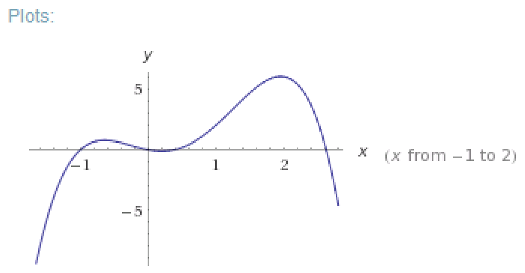
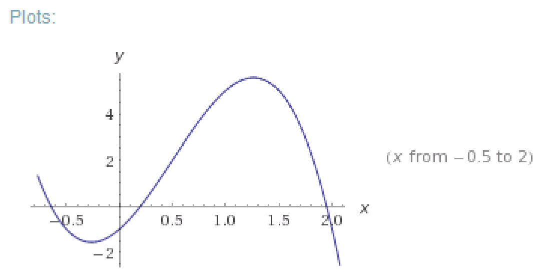
INF3490 exercise answers - week 1 2014

Problem 1

a) f(x) to the left, and f'(x) to the right.





b) Code:

```python
stepSize = 0.01
precision = 0.0001

def f(x):
    return -x**4 + 2*(x**3) + 2*(x**2) - x

def df(x):
    return -4*(x**3) + 6*(x**2) + 4*x - 1

def gradientAscent(f, df, start, step, prec):

    xNew = start
    xOld = start + 1 #just add something to start the while-loop
    while(abs(xNew - xOld) > prec):
        xOld = xNew
        xNew = xOld + step*df(xOld)

    print 'Local max found at: ', xNew, ', with value: ', f(xNew)

#Testing a few start points
gradientAscent(f, df, 3, stepSize, precision)
gradientAscent(f, df, -1, stepSize, precision)
gradientAscent(f, df, 0.20, stepSize, precision)
gradientAscent(f, df, 2, stepSize, precision)
```

ex1gradient.py

The starting point will affect the algorithm's performance. If the starting point is placed where the graph is ascending or descending to/from the local maximum at $x \approx -0.65$, the algorithm will end up on the local maximum.
Because $f'(x)$ is 0 in the local minimum at $x \approx -0.2$, the algorithm may not find a local maximum if our starting point is too close to the local minimum.

c) Code:

```python
stepSize = 0.5

def f(x):
    return -x**4 + 2*(x**3) + 2*(x**2) - x

def df(x):
    return -4*(x**3) + 6*(x**2) + 4*x - 1

def exhaustiveSearch(f, df, start, stop, step):
    x = start;
    max = start;
    while x < stop:
        if (f(x) > f(max)):
            max = x
        x = x + step

    print 'Max found at: ', max, ', with value: ', f(max)

#Running the algorithm
exhaustiveSearch(f, df, -2, 3, stepSize)
```

ex1exhaustive.py

When running the exhaustive search with step size 0.5 we find the global maximum at $x = 2.0$ and value 6.0. Try performing the search with a smaller step size to see if the result gets more accurate.

d) A greedy algorithm would check both directions and choose the best one, while a hill climber would choose one direction randomly and move if an improvement is found. The greedy algorithm would always find the global optimum when starting at $x = 0$, while the hill climber would have a 50-50 chance of reaching either the global or the local optimum from that point.

e Exhaustive search is the most efficient in this case. This is because the number if iterations needed to get from one end of the search space to the other is large in relative to the size of the search space. Simulated annealing needs to be able to jump pretty quickly back and forth in order to be efficient. In a search space of higher dimensionality simulated annealing would most likely outperform an exhaustive search.

f) Perhaps the easiest thing to do would be to add a random reset after a certain number of iterations. In that way the algorithm would over time be able to find several optima. Hill climbing, which has a certain randomness to it, could also benefit from back-tracking: When reaching some optima, it may back-track to some point it has visited earlier, and try a different direction from there.

Problem 2

a) The $(1 + 4)$ ES would generate four candidate solutions from the same origin before potentially changing parent instead of one. This would make for a more informed choice in which direction to move in the search space. The higher λ is, the more information is gathered about the neighborhood of the current solution. Recall that greedy search checks out all neighbors before making a move, while hill climbing on makes one. So we would expect the $(1 + \lambda)$ ES to behave increasingly like a greedy search as we increase λ.

b) An adaptive search strategy will, in most cases, increase the convergence rate of the search especially in the late stages. However, it does not by itself help avoid getting stuck in local optima.

c) When strategy parameters are mutated first, the change in strategy has immediate effect on the new solution that is created. Thus the fitness of this solution also indirectly rates the strategy to some degree. If the strategy parameters are mutated after the solution parameters this link is weaker, and we would expect the strategies to adapt slower, if at all.

Problem 3

a)

```python
import random
startPopulation = [1.0,2.0,3.0,4.0]

def g(x):
    return x

def ir (lop, noo):
    #noo number of offspring
    #lop = List of population
  length = len(lop)
    for i in range(0, noo):
        c=random.randint(0,length-1)
        d=random.randint(0,length-1)
        lop.append((lop[c]+lop[d])/2)
    print lop
    return lop

def os (lop, numberOfParents): #offspring selection
    newGeneration =[]
    for i in range(0, numberOfParents):
        mx = max(lop)
        newGeneration.append(mx)
        lop.remove(mx)
    print newGeneration
    return newGeneration

def es (population, numberOfparents, numberOfoffspring, iterations):
    for i in range (0, iterations):
        population =  ir(population, numberOfoffspring)
        population = os(population, numberOfParents)
    print population
  print 'Max found at: ', max(population)

# Running the algorithm
es (startPopulation, 4, 8, 3)
```

inf3490ex1P3a.py

b) Typical population development:

$\{4, 3, 2, 1\} \to \{4, 3.5, 3.5, 3\} \to \{4, 3.75, 3.75, 3.75\} \to \{4, 3.875, 3.75, 3.75\}$

The best solution will always survive as an offspring, so it just needs to be created at least once in the first place. If parents are drawn without replacement the probability is zero - 4 can only be one of the parents. Otherwise the probability of any one offspring having 4 as both parents is $\frac{1}{4} \cdot \frac{1}{4} = \frac{1}{16}$. The chance of none of the eight offspring having 4 as both parents is $\left(\frac{15}{16}\right)^8 \approx 0.597$ and so the probability of the solution surviving is $1 - 0.597 = 0.403$.

c) Typical population development:

$\{4,3,2,1\} \rightarrow \{4,3,2,3\} \rightarrow \{4,4,3,2\} \rightarrow \{4,4,4,3\}$

Although the mechanisms are quite different both algorithms move towards a population of only 4's (or, at least some close approximate of 4), as expected. We can see that while the ES is able to generate a number of new variants of the original solutions through recombination, the EP is completely dependent on its mutation operator to create anything new.

```
import random
startPopulation = [4,2,3,1]

def g(x):
    return x

def mutation(parents):
    length = len(parents)
    for i in range(0, length):
        c=random.randint(1,4)
        parents.append(c)
    print parents
    return parents

def ss (current): #survival selection
    newGeneration =[]
    length = len(current)
    for i in range(0, length/2):
        c=current[random.randint(0,length-1)]
        d=current[random.randint(0,length-1)]
        if c > d:
            newGeneration.append(c)
        else:
            newGeneration.append(d)
    print newGeneration
    return newGeneration

def ep (population, iterations):
    for i in range (0, iterations):
        population =  mutation(population)
        population = ss(population)
    mx = max(population)
    print 'Max found at: ', mx
    return max

# Running the algorithm
ep (startPopulation, 3)
```

INF3490ex1p3c.py