Lecture 10

Computational Programming in Matlab

In this lecture we discuss some classic algorithms that are the building blogs for many numerical problems. It is important to understand the principles behind those algorithms even if in practice one uses a build-in function or library.

Root finding

Problem:

Given a function $f(x)$ find the value of x_0 for that $f(x_0) = 0$ within an interval $a < x_0 < b$.

Assuming that the function is continuous and $f(a)$ and $f(b)$ are of opposite sign. In this case we can be sure that there is a root in the interval.

Linear Search

A simple approach for finding the root is **linear search**. We start at one end of the interval and check the function value for its its close to zero within a desired precision in x. Going in step sizes of the precision, we check if the sign of the function value changes to detect if we step over the zero.

```
f = @(x) ((x-2)^3-8);
a=-10;
b=10;
x0 = a;
nsteps = 0;
precision = 0.001;
while( (sign(f(a)) == sign(f(x0)))|| nsteps>1000000)
    x0 = x0+precision;
    nsteps = nsteps+1;
end
x0
```

```
x0 = 4.0010
```

```
f(x0)
```

```
ans = 0.0120
```

```
nsteps
```

```
nsteps = 14001
```

The linar algorithm is simple but takes many steps to reach the target. For a precision of 0.01 it took already 14000 steps. The maximum possible number of points to check N is the interval divided by the precision

```
N = (b-a)/precision
```

```
N = 20000
```

In the worst case scenario the algorithm would take all N steps to find the root. The algorithm run time scales linearly in the order of N, written as O(N).

Binary Search

The common technique to solve this types of search problems more efficiently is using a **binary search** strategy. The following algorithm succesively halfs the interval at its mid point depending on the function value.

```
f = @(x) ((x-2)^3-8);
a=-10;
b=10;
x0 = (a+b)/2;
nsteps = 0;
precision = 0.00000001;
a0=a;b0=b;
while(abs(b0-a0)>precision || nsteps>10000)
    if sign(f(x0))==sign(f(b0))
         b0 = x0;
    else
        a0 = x0;
    end
    x0 = (a0+b0)/2;
    nsteps = nsteps+1;
end
x0
```

```
x0 = 4.0000
```

```
f(x0)
```

```
ans = -1.1176e-08
```

```
nsteps
```

```
nsteps = 31
```

The strategy is called binary search because at each step we take n binary decisions that half the search interval, from N points, to N/2 points, to N/4, N/8 ... down to a single target point, this is with $N * \left(\frac{1}{2}\right)^n = 1$ the expected number of iterations is

$n = \log_2 N$

Comparing the expected runtime for a precision of 0.1 and 0.001 shows that the binary search algorithm only doubles the number of steps even if the number of search points increases by a factor of 100.

```
log2((b-a)/0.1)
```

```
ans = 7.6439
```

```
log2((b-a)/0.001)
```

```
ans = 14.2877
```

Newton Raphson Algorithm

One key aspect that determines the efficiency of the search algorithm is the strategy one uses to decide on the next point to check. In the case of linear search it was just taking the next point, in the case of binary search we jumped accross the search space to the mid point.

In strategy of the Newton Raphson Algorithm is to determine the candidate for the next search point using the linear expansion of the function f. Assuming we expand the function f around the current search point $x_n$we have to first order

$$f(x_{n+1}) = f(x_n) + f'(x_n)(x_{n+1} - x_n)$$

and desire that the next search point x_{n+1}leads directly to the root $f(x_{n+1}) = 0$, we get

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

In case that the function f is actually linear, this will lead us directly to the root. In case f is non-linear there will be an error and the procedure is repeated. The algorithm requires that one calculates the first derivative and the function value at each step.

```
f = @(x) ((x-2)^3-8);
df =  @(x) (3*(x-2)^2);
a=-10;
b=10;
x0 = a;
nsteps = 0;
precision = 0.001;
xnext = b;
while(nsteps<10000)
    xnext = x0-f(x0)/df(x0);
    nsteps = nsteps+1;
    if(abs(xnext-x0)<precision)
        x0=xnext;
        break
    end
    x0=xnext;
end
x0
```

```
x0 = 4.0000
```

```
f(x0)
```

```
ans = 5.4597e-06
```

```
nsteps
```

```
nsteps = 11
```

Because of its property of dividing the search space the runtime of the Newton Raphson algorithm also scales with log2(N) multiplied by the extra time if takes to calculate the derivatives.

Optimization

Optimization is another common numerical problem. In this case we are searching the maximum or minimum of a function. It is closely related to root finding, because the first derivative of a function is zero at its extreme points. Thus, one can formulate an optimization problem as a root finding problem of the derivative.

Newton Raphson Algorithm

The Newton Raphson algorithm applied to the optimization problem can be derived from the second order expansion

$$f(x_{+1}) = f(x_n) + f'(x_n)(x_{n+1} - x_n) + \frac{1}{2} f''(x_n)(x_{n+1} - x_n)^2$$

Taking the derivative with respect to x_{n+1} while holding $x_n$constant

$$f'(x_{+1}) = f'(x_n) + f''(x_n)(x_{n+1} - x_n)$$

Solving for the location of the extreme point $x_{n+1}$with $f'(x_{n+1}) = 0$ we find

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

which leads to the same type of algorithm for optimization as for root finding, only that we need to exchange the function f and its derivative by the first and second derivatives.

Below the use of the Newton Raphson Algorithm to find the minimum of a shifted cosh function

```
f = @(x) (cosh(x-2)+2);
df =  @(x)  (sinh(x-2));
ddf =  @(x) (cosh(x-2));
a=-1;
b=5;
xmin = a;
nsteps = 0;
precision = 0.0001;
xnext = b;
while(nsteps<10000)
    xnext = xmin-df(xmin)/ddf(xmin);
    nsteps = nsteps+1;
    if(abs(xnext-xmin)<precision)
        xmin=xnext;
        break
    end
    xmin=xnext;
end
xmin
```

```
xmin = 2
```

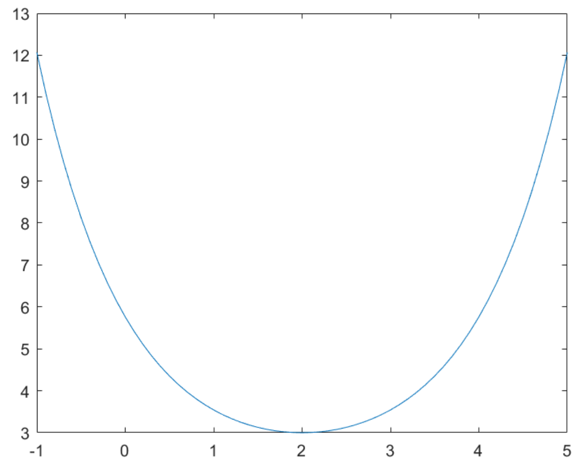
```
f(xmin)
```

```
ans = 3
```

```
nsteps
```

```
nsteps = 6
```

```
plot(a:.1:b,f(a:.1:b));
```



Gradient Descent

The gradient descent algorithm can be seen as a version of the Newton Raphson algorithm with the second derivative set to a constant value. It is used even in case the second derivative is not really a constant but just unknown or difficult to compute. The search point updating strategy is

$$x_{n+1} = x_n - \alpha\, f'(x_n)$$

where the constant α is sometimes called the learning rate and translates to a scaling of the step size by the inverse of the estimated second derivative. The value of alpha is important for the algorithm to converge. Here we need to choose a positive alpha, since we are looking for a minimum where the function is convex function, meaning it has a positive curvature.

```
f = @(x) (cosh(x-2)+2);
df =  @(x) (sinh(x-2));

xguess = -3;
alpha = .1;

nsteps = 0;
precision = 0.0001;
xvalues = [xguess];
while(nsteps<1000)
    xnext = xguess-alpha*df(xguess);
    nsteps = nsteps+1;
    if(abs(xnext-xguess)<precision)
        xguess=xnext;
        break
    end
    xguess=xnext;
    xvalues = [xvalues xguess];
end
xmin=xguess
```

```
xmin = 2.0009
```

```
nsteps
```

```
nsteps = 72
```

```
f(xmin)
```

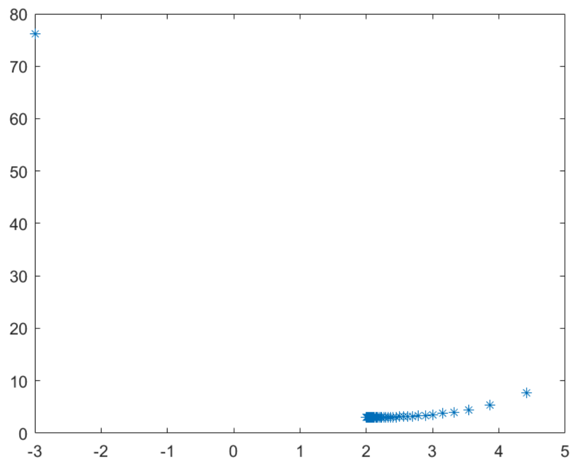
```
ans = 3.0000
```

We find that the algoritm above does not convert for starting point xguess=5 and alpha=1. In order to analyze the behaviour of the algorithm we record and plot the succesive values of x that are used in the search.

```
xvalues
```

```
xvalues = 1×72
   -3.0000    4.4203    3.8623    3.5481    3.3236    3.1491    3.0072    2.8885 ···
```

```
plot(xvalues,f(xvalues),'*')
```



Task 1

It turns out the algorithm is sensitive to settings of the starting search point xmin and alpha. The praxis of searching for a set of parameters the 'work' well is called **hyperparameter tuning**.

Try different values of alpha for a fixed starting point of xguess=5 and record the performance of the algorithm in the table below:

Alpha	Nsteps	Xmin	f(xmin)
1	1000	NaN	NaN
0.9			
0.8			
0.7			
0.6			
0.5			
0.4			
0.3			
0.2			
0.1			

Next vary the starting point xmin for a fixed choice of alpha = 0.2 and record the performance of the algorithm in the table below:

X0	Nsteps	Xmin	f(xmin)
−9	1000	NaN	NaN
−7			
−5			
−3			
−1			
1			
3			
5			
7			
9			

Gradient Descent in Vectorspace

The gradient descent generalizes to multiple dimensions. In this case we want to find the minimum of a function $f\left(\overrightarrow{x_{\min}}\right)$ where the search space consists of m dimensional vectors with components

$$\vec{x} = \begin{bmatrix} x_1 & x_2 & \dots & x_{m-1} & x_m \end{bmatrix}$$

Now the first derivative in the update rule generalizes to the gradient of the function f with respect to x

$$\vec{x}_{n+1} = \vec{x}_n - \alpha \nabla f(x_n)$$

This are m update equations, one for each component.

$$(x_1)_{n+1} = (x_1)_n - \alpha \frac{\partial}{\partial x_1} f\left(\vec{x}_n\right)$$

...

$$(x_m)_{n+1} = (x_m)_n - \alpha \frac{\partial}{\partial x_m} f\left(\vec{x}_n\right)$$

Task 2

Generalize the gradient decent algorithm to vector variables for the quadratic function f given below. Fill in the missing gradient df below:

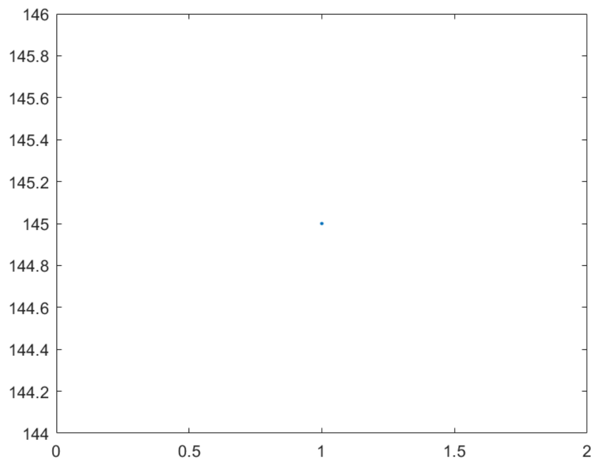
```
f = @(x) ( (x(1)-1)^2+(x(2)-2)^2);
df = @(x) ([0 ; 0 ]); % FILL IN THE GRADIENT HERE

xguess = [10;10];
alpha = .1;

nsteps = 0;
precision = 0.001;
xvalues = [xguess];
fvalues = [f(xguess)];
while(nsteps<1000)
    xnext = xguess; % MODIFY THE UPDATE STEP
    nsteps = nsteps+1;
    if(norm(xnext-xguess)<precision)
        xguess=xnext;
        break
    end
    xguess=xnext;
    xvalues = [xvalues xguess];
    fvalues = [fvalues f(xguess)];
end
xmin=xguess
```

```
xmin = 2×1
    10
    10
```

```
nsteps
```

```
nsteps = 1
```

```
plot(1:length(fvalues),fvalues,'.')
```



Solving a Regression Problem with Gradient Descent

In linear regression we try to determine a model the best fits to the data. As data we consider a matrix $\hat{D}$ where each row represents a vector of the 'independent' variable $\vec{d}$ and a vector of dependent variable measurements $\vec{t}$. The goal is to determine a model parameter vector $\vec{x}$ that best fits to the data.

Determine $\vec{x}$ to fit

$$\vec{t} = \hat{D}\vec{x}$$

or in components

$$t_i = \sum_j d_{\mathrm{ij}} x_j$$

For example, for a model that predicts the observed house prices given the number of square feed size and number of bedrooms.

$$\begin{bmatrix} \text{House Price 100th} \\ 2 \\ 3 \\ 4 \\ 4.2 \\ 5 \end{bmatrix} = \begin{bmatrix} \text{Sq Feed(1000th)} & \text{Bedrooms} & \text{Offset} \\ 1 & 1 & 1 \\ 1.5 & 1 & 1 \\ 2 & 2 & 1 \\ 2.2 & 2 & 1 \\ 2.5 & 3 & 1 \end{bmatrix} * \begin{bmatrix} \text{Model} \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

The error of the model with parameters $\vec{x}$ in fitting to the data is the differences between the actually obeserved data and the prediction.

$$\vec{e} = \left(\hat{D}\vec{x} - \vec{t}\right)$$

or again in components

$$e_i = \sum_j d_{\mathrm{ij}} x_j - t_i$$

To fit the regression we minimize the function f(x) of the squared error

$$f\left(\vec{x}\right) = \frac{1}{2}\sum_i e_i^2 = \frac{1}{2}\left(\hat{D}\vec{x} - \vec{t}\right)^2$$

$$= \frac{1}{2}\left(\hat{D}\vec{x} - \vec{t}\right)'\left(\hat{D}\vec{x} - \vec{t}\right) = \frac{1}{2}\left(\vec{x}'\hat{D}'\hat{D}\vec{x} - \vec{x}'\hat{D}'\vec{t} - \vec{t}'\hat{D}\vec{x} + \vec{t}'\vec{t}\right)$$

Its gradient is

$$\nabla f\left(\vec{x}\right) = \hat{D}'\hat{D}\vec{x} - \hat{D}'\vec{t} = \hat{D}'\left(\hat{D}\vec{x} - \vec{t}\right) = \hat{D}'\vec{e}$$

Task 3

Solve the linear regression problem above by adapting your algorithm from Task 2 to find the best x that explains the observed housing price vector. Search for best starting points xguess and alpha.

Starter code:

```
D = [[1 1 1];[1.5 1 1];[2 2 1];[2.2 2 1];[2.5 3 1];];
t=[2.5;3;4;4.2;5];
f = @(x)  (0); % FILL IN;
df = @(x) (0); % FILL IN;
xguess =[2;2;2];
alpha = .0000001;

nsteps = 0;
precision = 0.00001;
xvalues = [xguess];
fvalues = [f(xguess)];
while(nsteps<2000)
    xnext = xguess;   % MODIFY THE UPDATE STEP
    nsteps = nsteps+1;
    if(norm(xnext-xguess)<precision)
        xguess=xnext;
        break
    end
    xguess=xnext;
    xvalues = [xvalues xguess];
    fvalues = [fvalues f(xguess)];
end
```

```
xmin=xguess
```

```
xmin = 3×1
     2
     2
     2
```

```
nsteps
```

```
nsteps = 1
```

Check

```
df(xmin)
```

```
ans = 0
```

```
%xvalues
plot(10:length(fvalues),fvalues(10:length(fvalues)),'.')
```