Labor force participation

In this example, we evaluate which factors are important in determining the participation of women in the labor market. This example was borrowed from Greene's textbook Econometric Analysis available here (https://pages.stern.nyu.edu/~wgreene/Text/econometricanalysis.htm), using data from the Mroz (1987) study of the labor supply of married women. The end goal of this dataset is to estimate a wage equation for women using the Heckman Selection model (Heckman (1976) "The Common Structure of Statistical Models of Truncation, Sample Selection, and Limited Dependent Variables and a Simple Estimator for Such Models" Annals of Economic and Social Measurement, 5, 475-492.), but this will be introduced at the end for completeness (but not covered during the TA session).

The variables considered are:

1. Labor force participation (LFP)
2. Woman's age (WA)
3. Square of Woman's age (to be created)
4. Family income, in 1975 dollars (FAMINC)
5. Wife's educational attainment, in years (WE)
6. Number of children (KIDS, to be created as dummy variable 1, if the number is kids is higher than 0, or 0 if not (use KL6 and K618))

For those familiar with either Python, Matlab or Julia, find a cheatsheet on the website of quantecon here (https://cheatsheets.quantecon.org/)

In [1]:
```
# Load packages that will be useful below

# for dataframes, matrix algebra and graph respectively
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# for stats
from scipy import stats
import statsmodels.api as sm
import statsmodels.formula.api as smf

# for optim
from scipy.optimize import minimize
from numpy.linalg import norm



# for numerical errors
import sys
eps = sys.float_info.epsilon  ## smallest numerical error

#for tables
from tabulate import tabulate
```

```
/opt/anaconda3/lib/python3.7/site-packages/statsmodels/tools/_testing.py:19: FutureWarning: pandas.util.testing is deprecated. Use the functions in the public API at pa
ndas.testing instead.
  import pandas.util.testing as tm
```

For optimization routines, check the documentation of scipy

https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html (https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html)

1. Process and clean the data to created the 6 variables needed

In [2]:
```
#with open('TableF5-1.csv') as csvfile:
#    data_raw = csv.reader(csvfile, delimiter=',')

df = pd.read_csv ('TableF5-1.csv')

df['Intercept'] = 1
df['WA2'] = df.WA**2
df['KIDS'] = (df.KL6 + df.K618 > 0)*1

print(df)

# Processed as a matrix, to be used in the functions below

data_mat_y = np.array( df['LFP'])
##print(np.shape(data_mat_y)) # should be a 753 vector

data_mat_x =  np.array(df[['Intercept','WA', 'WA2', 'FAMINC', 'WE', 'KIDS']])
##print(np.shape(data_mat_x)) # should be a 753 vector
```

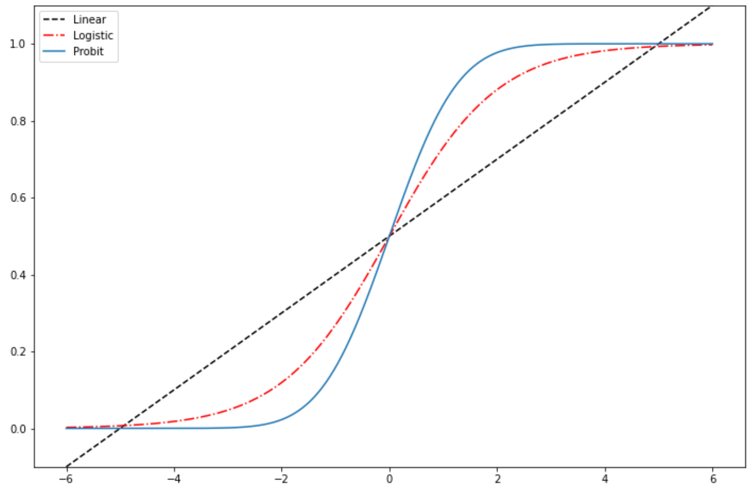
```
     LFP  WHRS  KL6  K618  WA  WE      WW  RPWG  HHRS  HA  ...  FAMINC  \
0      1  1610    1     0  32  12  3.3540  2.65  2708  34  ...   16310
1      1  1656    0     2  30  12  1.3889  2.65  2310  30  ...   21800
2      1  1980    1     3  35  12  4.5455  4.04  3072  40  ...   21040
3      1   456    0     3  34  12  1.0965  3.25  1920  53  ...    7300
4      1  1568    1     2  31  14  4.5918  3.60  2000  32  ...   27300
..   ...   ...  ...   ...  ..  ..     ...   ...   ...  ..  ...     ...
748    0     0    0     2  40  13  0.0000  0.00  3020  43  ...   28200
749    0     0    2     3  31  12  0.0000  0.00  2056  33  ...   10000
750    0     0    0     0  43  12  0.0000  0.00  2383  43  ...    9952
751    0     0    0     0  60  12  0.0000  0.00  1705  55  ...   24984
752    0     0    0     3  39   9  0.0000  0.00  3120  48  ...   28363

        MTR  WMED  WFED    UN  CIT  AX  Intercept   WA2  KIDS
0    0.7215    12     7   5.0    0  14          1  1024     1
1    0.6615     7     7  11.0    1   5          1   900     1
2    0.6915    12     7   5.0    0  15          1  1225     1
3    0.7815     7     7   5.0    0   6          1  1156     1
4    0.6215    12    14   9.5    1   7          1   961     1
..      ...   ...   ...   ...  ...  ..        ...   ...   ...
748  0.6215    10    10   9.5    1   5          1  1600     1
749  0.7715    12    12   7.5    0  14          1   961     1
750  0.7515    10     3   7.5    0   4          1  1849     0
751  0.6215    12    12  14.0    1  15          1  3600     0
752  0.6915     7     7  11.0    1  12          1  1521     1

[753 rows x 22 columns]
```

In [ ]:

2. Use preexisting packages to estimate the logit and probit parameters (glm in R, statsmodels in Python, fitglm for Matlab, glm for Julia)

In [3]:
```
fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111)
support = np.linspace(-6, 6, 1000)
ax.plot(support, 0.5+ 0.1*support, "k--", label="Linear")
ax.plot(support, stats.logistic.cdf(support), "r-.", label="Logistic")
ax.plot(support, stats.norm.cdf(support), label="Probit")
ax.set_ylim((-0.1,1.1))
ax.legend()
```

Out[3]:
```
<matplotlib.legend.Legend at 0x7fedb7e26310>
```



In [4]:
```
#formula = "LFP ~ WA + WA2 + FAMINC + WE + KIDS"
```

Linear probability model

$$p = \mathbb{P}(X = 1) = X'\beta$$

In [5]:
```
lpm_mod = sm.OLS(df['LFP'], df[['Intercept','WA', 'WA2', 'FAMINC', 'WE', 'KIDS']])
lpm_res = lpm_mod.fit()
print("Parameters: \n", lpm_res.params)
print("\n  t-values \n", lpm_res.tvalues,"\n  p-values \n",lpm_res.pvalues)
```

```
Parameters:
 Intercept   -1.027963
WA           0.068300
WA2         -0.000893
FAMINC       0.000002
WE           0.035730
KIDS        -0.159947
dtype: float64

  t-values
 Intercept   -1.968831
WA           2.771059
WA2         -3.097605
FAMINC       1.046135
WE           4.267616
KIDS        -3.379964
dtype: float64
  p-values
 Intercept    0.049342
WA           0.005726
WA2          0.002024
FAMINC       0.295837
WE           0.000022
KIDS         0.000763
dtype: float64
```

Logit model

$$p = \mathbb{P}(X = 1) = \sigma(X'\beta) = \frac{1}{1 + e^{-X'\beta}}$$

In [6]:
```
logit_mod = sm.Logit(df['LFP'], df[['Intercept', 'WA', 'WA2', 'FAMINC', 'WE', 'KIDS']])
logit_res = logit_mod.fit(disp=0)
print("Parameters: \n", logit_res.params)
print("\n  t-values \n", logit_res.tvalues,"\n  \n p-values \n",logit_res.pvalues)
```

```
Parameters:
 Intercept   -6.646031
WA           0.296175
WA2         -0.003881
FAMINC       0.000008
WE           0.157328
KIDS        -0.720503
dtype: float64

  t-values
 Intercept   -2.903332
WA           2.745193
WA2         -3.064816
FAMINC       1.128550
WE           4.170993
KIDS        -3.361028
dtype: float64

 p-values
 Intercept    0.003692
WA           0.006048
WA2          0.002178
FAMINC       0.259088
WE           0.000030
KIDS         0.000777
dtype: float64
```

Probit model

$$p = \mathbb{P}(X=1) = \Phi(X'\beta) = \int_{-\infty}^{X'\beta} \phi(x)dx$$

with Φ the c.d.f and ϕ the p.d.f of the Normal distribution

In [7]:
```
probit_mod = sm.Probit(df['LFP'], df[['Intercept', 'WA', 'WA2', 'FAMINC', 'WE', 'KIDS']])
probit_res = probit_mod.fit(disp=0)
print("Parameters: \n", probit_res.params)
print("\n  t-values \n", probit_res.tvalues,"\n  \n p-values \n",probit_res.pvalues)
```

```
Parameters:
 Intercept   -4.156807
WA           0.185395
WA2         -0.002426
FAMINC       0.000005
WE           0.098182
KIDS        -0.448987
dtype: float64

  t-values
 Intercept   -2.964730
WA           2.810436
WA2         -3.136096
FAMINC       1.088918
WE           4.271744
KIDS        -3.429697
dtype: float64

 p-values
 Intercept    0.003029
WA           0.004947
WA2          0.001712
FAMINC       0.276190
WE           0.000019
KIDS         0.000604
dtype: float64
```

3. Create a function for the log-likelihood of the logit and probit models, use an optimization routine to minimize this function numerically

$$\ell(\theta|\{y,x\}) = \sum_{i=1}^{n} y_i \log(p(x_i)) + (1-y_i)\log(1-p(x_i))$$
$$p(x_i) = \sigma(X_i'\beta) \qquad [logit]$$
$$p(x_i) = \Phi(X_i'\beta) \qquad [probit]$$

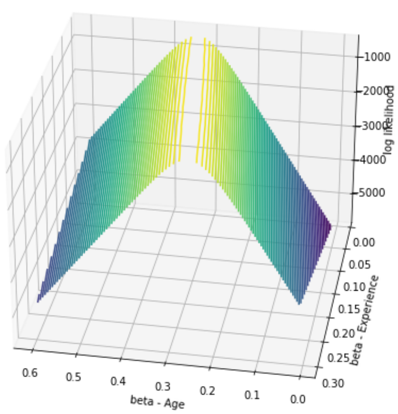
In [8]:
```
def loglik_logit(beta):
    linprob = data_mat_x @ beta
    prob_i = stats.logistic.cdf(linprob)
#    if np.min(prob_i<=0) or np.max(prob_i>=1):
#        print("Error proba >1 or <0")
#        print(beta)
## adding epsilon inside the log, because numerical rounding often gives prob_i = 0 or 1
    loglik = sum( data_mat_y * np.log(prob_i + eps) + (1-data_mat_y) * np.log(1 - prob_i + eps))
    return loglik

def loglik_probit(beta):
    linprob = data_mat_x @ beta
    prob_i = stats.norm.cdf(linprob)
    loglik = sum( data_mat_y * np.log(prob_i + eps) + (1-data_mat_y) * np.log(1 - prob_i + eps))
    return loglik
```

In [9]:
```
loglik_logit(logit_res.params)
```

Out[9]:
```
-490.9838664567777
```

In [10]:
```
#plotting the log likeihood in 2d

def loglik_logit_plot_2par(beta1,beta2):
    betaconcat=np.array([logit_res.params[0], beta1, logit_res.params[2],logit_res.params[3],beta2, logit_res.params[5]])
    return loglik_logit(betaconcat)


beta1_vec = np.linspace(0.01,0.6, 50)
beta2_vec = np.linspace(0.01,0.3, 50)

beta1_mesh, beta2_mesh = np.meshgrid(beta1_vec, beta2_vec)
loglik_mesh = np.empty((50, 50))

for i in range(0, 50):
    for j in range(0,50):
        loglik_mesh[i,j] = loglik_logit_plot_2par(beta1_mesh[i,j], beta2_mesh[i,j])
```

In [11]:
```
fig = plt.figure(figsize=(16,8))
ax = plt.axes(projection='3d')
ax.contour3D(beta1_mesh, beta2_mesh, loglik_mesh, 80)#, cmap='binary')
ax.set_xlabel('beta - Age')
ax.set_ylabel('beta - Experience')
ax.set_zlabel('log likelihood');
ax.view_init(azim = 100, elev=30.) ;

## we see that there are many values (couples beta_age and beta_we) that can be considered as local maxima.
## likely to be even worse in 6 dimensions
```



In [45]:
```
beta_init = 0.5*(logit_res.params + probit_res.params) + np.array([1.0,0.1,0.001,0.0001, 0.1, 0.2])*np.random.randn(6)
beta_init = np.array(beta_init)
#beta_init = 0.5*(logit_res.params + probit_res.params)
# average of the guess for probit/logit + random noise
# multiply with variance (small enough) that's roughly proportional to the value of the parameter considered.
```

In [46]:
```
# convert the function for minimization

def loglik_logit_min(beta):
    loglik = - loglik_logit(beta)
    return loglik

def loglik_probit_min(beta):
    loglik = - loglik_probit(beta)
    return loglik
```

In [47]:
```
## Using non-gradient method:
# Nelder Mead
res_nm_logit = minimize(loglik_logit_min, beta_init, method='Nelder-Mead', tol=1e-6)
res_nm_probit = minimize(loglik_probit_min, beta_init, method='Nelder-Mead', tol=1e-6)
```

In [48]:
```
print(res_nm_logit.x)
print(res_nm_probit.x)
```

```
[-6.64603040e+00  2.96175214e-01 -3.88138168e-03  8.01040495e-06
  1.57328062e-01 -7.20503007e-01]
[-4.15680664e+00  1.85395087e-01 -2.42589697e-03  4.58044304e-06
  9.81822890e-02 -4.48986767e-01]
```

4. Create a function for the gradient of the log-likelihood (derivative w.r.t. to parameters c.f. analytical formula above), use that as an input in optimizations routines

In [166]:
```python
# already converted for minimization

def loglik_logit_grad(beta):
    linprob = data_mat_x @ beta
    prob_i = stats.logistic.cdf(linprob)
    gradloglik = - (data_mat_y - prob_i) @ data_mat_x   #no need for sum signs as the dot product give the appropriate dim
    return gradloglik


def loglik_probit_grad(beta):
    linprob = data_mat_x @ beta
    prob_i = stats.norm.cdf(linprob)
    derivprob_i = stats.norm.pdf(linprob)
    gradloglik = - ( (derivprob_i / (prob_i * (1- prob_i + eps))) * (data_mat_y - prob_i) ) @ data_mat_x
    return  gradloglik
```

In [50]:
```python
#loglik_logit_grad(res_nm_logit.x)
```

In [51]:
```python
# print initial conditions
print(logit_res.params)

print(beta_init)
```

```
Intercept   -6.646031
WA           0.296175
WA2         -0.003881
FAMINC       0.000008
WE           0.157328
KIDS        -0.720503
dtype: float64
[-5.61098601e+00  2.61988245e-01 -5.42849680e-03  5.32443483e-05
  2.37744185e-01 -5.98505639e-01]
```

In [167]:
```python
## conjugate gradient

res_cg_logit = minimize(fun = loglik_logit_min, x0 = beta_init, jac = loglik_logit_grad, method='CG', tol=1e-6)

res_cg_probit = minimize(fun = loglik_probit_min, x0 = beta_init, jac = loglik_probit_grad, method='CG', tol=1e-6)
```

In [53]:
```python
print(res_cg_logit.x)
print(res_cg_probit.x)
```

```
[-5.63425860e+00  2.49548886e-01 -3.34487015e-03  8.45169452e-06
  1.52818211e-01 -7.04656644e-01]
[-5.61098601e+00  2.61988364e-01 -5.42258334e-03  9.88383677e-05
  2.37744213e-01 -5.98505638e-01]
```

In [ ]:

In [488]:
```python
#plt.scatter(data_mat_x @ probit_res.params, stats.logistic.cdf(data_mat_x @ probit_res.params))
```

5. Use gradient descent methods (manually since you computed the gradient!) to minimize the likelihood (c.f. section 3.1. but no need to use the optimal learning rate)

In [198]:
```python
#delta = 1.0
tol_gd = 1e-7
maxiter = 30000


def grad_descent_logit(betainit, alpha):
    beta_old = betainit ;
    beta_seq = np.empty((np.shape(beta_init)[0], maxiter))
    grad_seq = np.empty((np.shape(beta_init)[0], maxiter))

    for it in range(0, maxiter):
        # always set a maximum number of iteration, otherwise the algo can run forever!
        # compute gradient
        beta_seq[:,it] = beta_old ;
        gradloglik_logit = loglik_logit_grad(beta_old)
        grad_seq[:,it] = gradloglik_logit

        # update the minimizing sequence:
        beta_new = beta_old - alpha * gradloglik_logit
        delta = beta_new - beta_old
        beta_old = beta_new
        if sum(abs(delta)) < tol_gd:
            print("Logit Gradient descent converged! :) iter : ", it)
            break
            # break the loop and exit!
        elif (it == maxiter-1):
            print("Gradient descent not converged! :'(", it)
    return beta_new, beta_seq, grad_seq


tol_gd = 1e-7

def grad_descent_probit(betainit, alpha):
    delta = 1.0
    beta_old = betainit ;
    beta_seq = np.empty((np.shape(beta_init)[0], maxiter))
    grad_seq = np.empty((np.shape(beta_init)[0], maxiter))

    for it in range(0, maxiter):
        # always set a maximum number of iteration, otherwise the algo can run forever!
        beta_seq[:,it] = beta_old ;

        # compute gradient
        gradloglik_probit = loglik_probit_grad(beta_old)
        grad_seq[:,it] = gradloglik_probit

        # update the minimizing sequence:
        beta_new = beta_old - alpha * gradloglik_probit
        delta = beta_new - beta_old
        beta_old = beta_new
        if sum(abs(delta)) < tol_gd:
            print("Probit Gradient descent converged! :) iter : ", it)
            break
            # break the loop and exit!
        elif (it == maxiter-1):
            print("Gradient descent not converged! :'(", it)
    return beta_new, beta_seq, grad_seq
```

In [ ]:

In [199]:
```python
loglik_probit_grad(beta_init)
alpha_test = 1e-12
```

In [200]:
```python
beta_gd_logit,beta_seq_logit, grad_seq_logit = grad_descent_logit(beta_init, alpha_test)
beta_gd_probit,beta_seq_probit,grad_seq_probit = grad_descent_probit(beta_init, alpha_test)
```

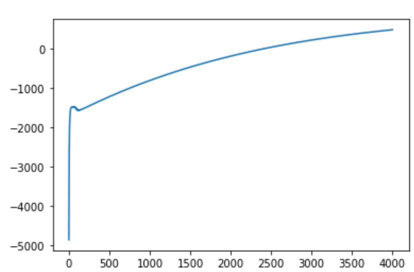
```
Logit Gradient descent converged! :) iter :  7168
Probit Gradient descent converged! :) iter :  4514
```

In [172]:
```python
beta_gd_probit
```

Out[172]:
```
array([-5.61098580e+00,  2.62009039e-01, -4.00616394e-03, -1.17768356e-05,
        2.37745232e-01, -5.98505890e-01])
```

In [369]:
```python
#gradient over time (it overshoots a little...)

plt.plot(range(0, 4000),grad_seq_probit[4,0:4000])
```

Out[369]:
```
[<matplotlib.lines.Line2D at 0x7feda4b5e4d0>]
```



In [ ]:

6. Use Newton-Raphson methods to find the zero of the First order conditions -- for that, you would need to compute numerically (or with automatic differentiation) the Hessian of the function, (i.e. the gradient of the gradient)

6.1 Newton Methods: with automatic differentiation

In [162]:
```
pip install autograd
```

```
Collecting autograd
  Downloading https://files.pythonhosted.org/packages/d9/6e/5aec16d68bf07e17e1a6cac5011e1c8f5f8dadb0ac5e875d432ee8aaa733/autograd-1.4-py3-none-any.whl (https://files.pythonhosted.org/packages/d9/6e/5aec16d68bf07e17e1a6cac5011e1c8f5f8dadb0ac5e875d432ee8aaa733/autograd-1.4-py3-none-any.whl) (48kB)
     |████████████████████████████████| 51kB 8.8MB/s eta 0:00:011
Requirement already satisfied: future>=0.15.2 in /opt/anaconda3/lib/python3.7/site-packages (from autograd) (0.18.2)
Requirement already satisfied: numpy>=1.12 in /opt/anaconda3/lib/python3.7/site-packages (from autograd) (1.17.2)
Installing collected packages: autograd
Successfully installed autograd-1.4
Note: you may need to restart the kernel to use updated packages.
```

In [218]:
```
# for automatic differentiation
import autograd.numpy as np  # Thinly-wrapped numpy
from autograd import jacobian
```

In [223]:
```
## Need to rewrite the function with function coded manually (instead of the packages stats.logistic and stats.norm)
# for that reason I'm only going to do the

def loglik_logit_grad_new(beta):
    linprob = data_mat_x @ beta
    prob_i = 1 / (1 + np.exp(- linprob))
    gradloglik = - (data_mat_y - prob_i) @ data_mat_x   #no need for sum signs as the dot product give the appropriate dim
    return gradloglik
```

In [220]:
```
loglik_logit_hess_ad = jacobian(loglik_logit_grad)
```

In [221]:
```
# does the hessian computed with Auto-diff works?
loglik_logit_hess_ad(beta_init)
# yes :D :D
```

Out[221]:
```
array([[1.18289124e+02, 4.70213377e+03, 1.91933710e+05, 2.72480328e+06,
        1.47488236e+03, 9.49883325e+01],
       [4.70213377e+03, 1.91933710e+05, 8.03841200e+06, 1.11357419e+08,
        5.88787417e+04, 3.64132992e+03],
       [1.91933710e+05, 8.03841200e+06, 3.44966820e+08, 4.66969747e+09,
        2.41426143e+06, 1.42622475e+05],
       [2.72480328e+06, 1.11357419e+08, 4.66969747e+09, 7.68141860e+10,
        3.47200392e+07, 2.14221705e+06],
       [1.47488236e+03, 5.88787417e+04, 2.41426143e+06, 3.47200392e+07,
        1.88709796e+04, 1.17984493e+03],
       [9.49883325e+01, 3.64132992e+03, 1.42622475e+05, 2.14221705e+06,
        1.17984493e+03, 9.49883325e+01]])
```

In [355]:
```
#delta = 1.0
tol_gd = 1e-6
maxiter = 5000
alpha_nr = 0.02
def newtonraphson_logit(betainit, alpha):
    beta_old = betainit ;
    beta_seq = np.empty((np.shape(beta_init)[0], maxiter))
    grad_seq = np.empty((np.shape(beta_init)[0], maxiter))

    for it in range(0, maxiter):
        # always set a maximum number of iteration, otherwise the algo can run forever!
        # compute gradient

        gradloglik_logit = - loglik_logit_grad(beta_old)
        hessloglik_logit = loglik_logit_hess_ad(beta_old)

        ## solve the linear system (c.f. page 6)           H(f)(x) (x_new - x_old) = - grad f
        delta_x = np.linalg.solve(hessloglik_logit, gradloglik_logit)

        # save stuffs
        grad_seq[:,it] = delta_x
        beta_seq[:,it] = beta_old ;

        # update the minimizing sequence: since delta_x = x_new - x_old
        beta_new =  beta_old + delta_x

        # use relaxation to converge smootly
        beta_new =  (1-alpha)* beta_old + alpha* beta_new

        beta_old = beta_new
        if sum(abs(delta_x)) < tol_gd:
            print("Logit Newton Raphson converged! :) iter : ", it)
            break
            # break the loop and exit!
        elif (it == maxiter-1):
            print("Gradient descent not converged! :'(", it)
    return beta_new, beta_seq, grad_seq
```

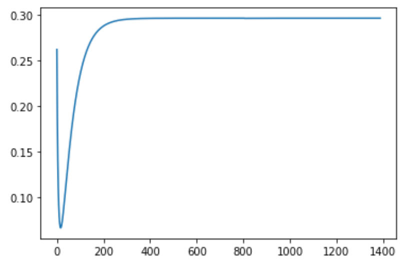
In [357]:
```
beta_nr_logit,beta_nr_seq_logit, grad_nr_seq_logit = newtonraphson_logit(beta_init, alpha_nr)
```

```
Logit Newton Raphson converged! :) iter :  804
```

In [ ]:

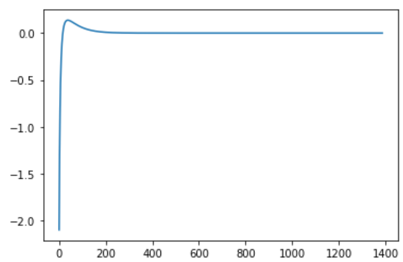
In [361]:
```
# plot beta (parameter values) over iterations

plt.plot(range(0,maxiter),beta_nr_seq_logit[1,:])
```

Out[361]: [<matplotlib.lines.Line2D at 0x7feda4974c50>]



In [362]:
```
# plot delta_x over iterations

plt.plot(range(0,maxiter),grad_nr_seq_logit[1,:])
```

Out[362]: [<matplotlib.lines.Line2D at 0x7feda4af33d0>]



6.2 Quasi Newton Methods: BFGS

In [224]:
```
## BFGS gradient

res_bfgs_logit = minimize(fun = loglik_logit_min, x0 = beta_init, jac = loglik_logit_grad, method='BFGS', tol=1e-6)

res_bfgs_probit = minimize(fun = loglik_probit_min, x0 = beta_init, jac = loglik_probit_grad, method='BFGS', tol=1e-6)
```

In [177]:
```
print(res_bfgs_logit.x)
print(res_bfgs_probit.x)
```

```
[-6.64603091e+00  2.96175240e-01 -3.88138199e-03  8.01040553e-06
  1.57328064e-01 -7.20503071e-01]
[-4.15680695e+00  1.85395096e-01 -2.42589700e-03  4.58044518e-06
  9.81822837e-02 -4.48986730e-01]
```

7. Did the different estimations of these parameters yield similar results, and if not, what could be a potential reason?

In [ ]:

In [366]:
```
df_results1_logit = pd.DataFrame(
np.array([np.array(logit_res.params),
res_nm_logit.x,
beta_gd_logit,
res_cg_logit.x,
beta_nr_logit,
res_bfgs_logit.x]),
index=[1, 2, 3,4,5, 6],
columns=['Intercept', 'WA', 'WA2', 'FAMINC', 'WE', 'KIDS'])

df_results0_logit = pd.DataFrame(
np.array(['GLM','NM', 'GD', 'CGD', 'NR', 'BFGS']),
index=[1, 2, 3,4,5,6],
columns=['Method'])

df_results2_logit = pd.DataFrame(
np.array([0,res_nm_logit.nit, 7168, res_cg_logit.nit, 804, res_bfgs_logit.nit]),
index=[1, 2, 3,4,5,6],
columns=['Nb iter'])


df_results_logit = pd.concat([df_results0_logit,df_results1_logit, df_results2_logit], axis=1)
```

In [367]:
```
df_results_logit
```
Out[367]:

	Method	Intercept	WA	WA2	FAMINC	WE	KIDS	Nb iter
1	GLM	-6.646031	0.296175	-0.003881	0.000008	0.157328	-0.720503	0
2	NM	-6.646030	0.296175	-0.003881	0.000008	0.157328	-0.720503	682
3	GD	-5.610986	0.262004	-0.004445	0.000023	0.237746	-0.598506	7168
4	CGD	-5.634259	0.249549	-0.003345	0.000008	0.152818	-0.704657	1200
5	NR	-6.646030	0.296175	-0.003881	0.000008	0.157328	-0.720503	804
6	BFGS	-6.646031	0.296175	-0.003881	0.000008	0.157328	-0.720503	19

In [ ]:

In [182]:
```
df_results1_probit = pd.DataFrame(
np.array([np.array(probit_res.params),
res_nm_probit.x,
beta_gd_probit,
res_cg_probit.x,
res_bfgs_probit.x]),
index=[1, 2, 3,4,5],
columns=['Intercept', 'WA', 'WA2', 'FAMINC', 'WE', 'KIDS'])

df_results0_probit = pd.DataFrame(
np.array(['GLM','NM', 'GD', 'CGD', 'BFGS']),
index=[1, 2, 3,4,5],
columns=['Method'])

df_results2_probit = pd.DataFrame(
np.array([0,res_nm_probit.nit, 4514, res_cg_probit.nit, res_bfgs_probit.nit]),
index=[1, 2, 3,4,5],
columns=['Nb iter'])


df_results_probit = pd.concat([df_results0_probit,df_results1_probit, df_results2_probit], axis=1)
```

In [368]:
```
# newton raphson with automatic differentiation for the hessian not computed
#(as the normal distribution c.d.f. requires to be coded up manually to work with auto-diff package)

df_results_probit
```
Out[368]:

	Method	Intercept	WA	WA2	FAMINC	WE	KIDS	Nb iter
1	GLM	-4.156807	0.185395	-0.002426	0.000005	0.098182	-0.448987	0
2	NM	-4.156807	0.185395	-0.002426	0.000005	0.098182	-0.448987	695
3	GD	-5.610986	0.262009	-0.004006	-0.000012	0.237745	-0.598506	4514
4	CGD	-5.612848	0.224560	-0.002825	0.000006	0.147281	-0.598021	41
5	BFGS	-4.156807	0.185395	-0.002426	0.000005	0.098182	-0.448987	16

In [ ]: