For the parts that require coding, you may use any software or programming languages you like but please present your source codes and results in a way that is comprehensible to someone who is unfamiliar with that program (e.g., comment your codes appropriately, present your results using tables and graphs). I recommend using NumPy, Mathematica, Matlab, R so that you don't have to code things from scratch.

1. Consider the function $f : \mathbb{R}^2 \to \mathbb{R}$ defined by

$$f(x, y) = \frac{1}{2}(ax^2 + by^2)$$

where $a, b > 0$. We will apply steepest descent with exact line search to f with the initial point $\mathbf{x}_0 = (x_0, y_0) = (b, a)$. (Note: In case it is not clear, you are supposed to do this problem 'by hand'. No coding required.)

(a) Show that f is strongly convex on $\mathbb{R}^n$. Find the global minimizer $\mathbf{x}_*$ and global minimum $f(\mathbf{x}_*)$.

Let $A = \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}; \mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}$. Then, $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\intercal A\mathbf{x}, \nabla f(\mathbf{x}) = A\mathbf{x}$ and $\nabla^2 f(\mathbf{x}) = A$.

So for $\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}, \mathbf{v}^\intercal \nabla^2 f(\mathbf{x})\mathbf{v} = \mathbf{v}^\intercal A\mathbf{v} = av_1^2 + bv_2^2 > \min(a, b)(v_1^2 + v_2^2)$, since $a, b > 0$.

Therefore, $\nabla^2 f(\mathbf{x}) \succ \min(a, b) \cdot I_2 \succ 0$ and $f(\mathbf{x})$ is strongly convex.

Let $\nabla f(\mathbf{x}) = A\mathbf{x} = \mathbf{0}$. Then, $\mathbf{x}_* = \mathbf{0}$ is the global minimizer and $f(\mathbf{0}) = \frac{1}{2} \cdot \mathbf{0} \cdot A \cdot \mathbf{0} = \mathbf{0}$ is the global minimum.

(b) Show that steepest descent with exact line search will yield step size

$$\alpha_k = \frac{2}{a + b},$$

iterates

$$\mathbf{x}_k = \begin{bmatrix} x_k \\ y_k \end{bmatrix}, \qquad x_k = \left(\frac{b - a}{a + b}\right)^k b, \qquad y_k = \left(\frac{a - b}{a + b}\right)^k a,$$

and function values

$$f(\mathbf{x}_k) = \frac{ab^2 + ba^2}{2}\left(\frac{b - a}{a + b}\right)^{2k}$$

for all $k \in \mathbb{N}$.

We will proceed by induction.

- Base step: $\mathbf{x}_0 = \begin{bmatrix} b \\ a \end{bmatrix}$. Our steepest descent update is the following:

$$\mathbf{x}_1 = \mathbf{x}_0 - \alpha_0 \nabla f(\mathbf{x}_0) = \mathbf{x}_0 - \alpha_0 A\mathbf{x}_0 = (I - \alpha_0 A)\mathbf{x}_0.$$

Then, we want to minimize

$$f(\mathbf{x_1}) = \frac{1}{2}\mathbf{x}_0^\intercal (I - \alpha_0 A)A(I - \alpha_0 A)\mathbf{x}_0 = \mathbf{x}_0^\intercal A\mathbf{x}_0 - 2\alpha_0 \mathbf{x}_0^\intercal A^2\mathbf{x}_0 + \alpha_0^2 \mathbf{x}_0^\intercal A^3\mathbf{x}_0.$$

Which is the same as minimizing $\alpha_0^2 \mathbf{x}_0^\top A^3 \mathbf{x}_0 - 2\alpha_0 \mathbf{x}_0^\top A^2 \mathbf{x}_0 + \frac{(\alpha_0 \mathbf{x}_0^\top A^2 \mathbf{x}_0)^2}{\mathbf{x}_0^\top A^3 \mathbf{x}_0}$ or

$$\alpha_0^2 - 2\alpha_0 \frac{\mathbf{x}_0^\top A^2 \mathbf{x}_0}{\mathbf{x}_0^\top A^3 \mathbf{x}_0} + \left(\frac{\alpha_0 \mathbf{x}_0^\top A^2 \mathbf{x}_0}{\mathbf{x}_0^\top A^3 \mathbf{x}_0} \right)^2 = \left(\alpha_0 - \frac{\mathbf{x}_0^\top A^2 \mathbf{x}_0}{\mathbf{x}_0^\top A^3 \mathbf{x}_0} \right)^2,$$

using addition and multiplication of positive constants. The value of α_0 that minimizes this is

$$\alpha_0 = \frac{\mathbf{x}_0^\top A^2 \mathbf{x}_0}{\mathbf{x}_0^\top A^3 \mathbf{x}_0} = \frac{a^2 b^2 + a^2 b^2}{a^3 b^2 + a^2 b^3} = \frac{1+1}{a+b} = \frac{2}{a+b},$$

as required. Additionally,

$$\mathbf{x}_1 = \left(I - \frac{2}{a+b} A \right) \mathbf{x}_0 = \begin{bmatrix} 1 - \frac{2a}{a+b} & 0 \\ 0 & 1 - \frac{2b}{a+b} \end{bmatrix} \begin{bmatrix} b \\ a \end{bmatrix},$$

and thus $x_1 = b\left(1 - \frac{2a}{a+b}\right) = b\left(\frac{a+b-a}{a+b}\right) = \left(\frac{b-a}{a+b}\right)^1 b.$

Similarly, $y_1 = a\left(1 - \frac{2b}{a+b}\right) = \left(\frac{a-b}{a+b}\right)^1 a.$

Finally, $f(\mathbf{x}_1) = \frac{1}{2}\left(a\left(\frac{b-a}{a+b}\right)^2 b^2 + b\left(\frac{a-b}{a+b}\right)^2 a^2\right) = \frac{1}{2}\left(\frac{b-a}{a+b}\right)^2 (ab^2 + a^2 b).$

- Induction step: Assume that for $\mathbf{x}_k$, steepest descent with exact line search will yield step size $\alpha_k = \frac{2}{a+b}$, iterates $x_k = \left(\frac{b-a}{a+b}\right)^k b, y_k = \left(\frac{a-b}{a+b}\right)^k a$ and function values $f(\mathbf{x}_k) = \frac{ab^2+ba^2}{2}\left(\frac{b-a}{a+b}\right)^{2k}$. We want to show that for $\mathbf{x}_{k+1}$, steepest descent with exact line search will yield step size $\alpha_{k+1} = \frac{2}{a+b}$, iterates $x_{k+1} = \left(\frac{b-a}{a+b}\right)^{k+1} b$ and $y_{k+1} = \left(\frac{a-b}{a+b}\right)^{k+1} a$ and function values $f(\mathbf{x}_{k+1}) = \frac{ab^2+ba^2}{2}\left(\frac{b-a}{a+b}\right)^{2(k+1)}$.
  We have $\mathbf{x}_{k+1} = (I - \alpha_k A)\mathbf{x}_k$. As we proved in the base case, minimizing $f(\mathbf{x}_{k+1})$ with respect to α_k results in the minimizer

$$\alpha_k = \frac{\mathbf{x}_k^\top A^2 \mathbf{x}_k}{\mathbf{x}_k^\top A^3 \mathbf{x}_k} = \frac{x_k^2 a^2 + y_k^2 a^2}{x_k^2 a^3 + y_k^2 b^3} = \frac{\left(\frac{b-a}{a+b}\right)^{2k} b^2 a^2 + \left(\frac{a-b}{a+b}\right)^{2k} a^2 b^2}{\left(\frac{b-a}{a+b}\right)^{2k} b^2 a^3 + \left(\frac{a-b}{a+b}\right)^{2k} a^2 b^3} = \frac{2}{a+b},$$

  as desired (since $(a-b)^2 = (b-a)^2$). Then,

$$\begin{aligned} \mathbf{x}_{k+1} &= (I - \alpha_k A)\mathbf{x}_k \\ &= \begin{bmatrix} 1 - \frac{2a}{a+b} & 0 \\ 0 & 1 - \frac{2b}{a+b} \end{bmatrix} \begin{bmatrix} \left(\frac{b-a}{a+b}\right)^k b \\ \left(\frac{a-b}{a+b}\right)^k a \end{bmatrix} \\ &= \begin{bmatrix} \frac{b-a}{a+b} & 0 \\ 0 & \frac{a-b}{a+b} \end{bmatrix} \begin{bmatrix} \left(\frac{b-a}{a+b}\right)^k b \\ \left(\frac{a-b}{a+b}\right)^k a \end{bmatrix} = \begin{bmatrix} \left(\frac{b-a}{a+b}\right)^{k+1} b \\ \left(\frac{a-b}{a+b}\right)^{k+1} a \end{bmatrix}, \end{aligned}$$

  and $f(\mathbf{x}_{k+1}) = \frac{1}{2}\left(a\left(\frac{b-a}{a+b}\right)^{2(k+1)} b^2 + b\left(\frac{a-b}{a+b}\right)^{2(k+1)} a^2\right) = \frac{ab^2+a^2b}{2}\left(\frac{b-a}{a+b}\right)^{2(k+1)}$, as required. □

(c) Deduce that

$$\lim_{k\to\infty} \frac{\|\mathbf{x}_{k+1} - \mathbf{x}_*\|}{\|\mathbf{x}_k - \mathbf{x}_*\|} = \left|\frac{a-b}{a+b}\right|, \qquad \lim_{k\to\infty} \frac{|f(\mathbf{x}_{k+1}) - f(\mathbf{x}_*)|}{|f(\mathbf{x}_k) - f(\mathbf{x}_*)|} = \left|\frac{a-b}{a+b}\right|^2$$

where $\|\cdot\|$ denotes the 2-norm. In other words, in this case the sequence of iterates is linearly convergent both in the usual sense ($\lim_{k\to\infty} \mathbf{x}_k = \mathbf{x}_*$ linearly) and the functional sense ($\lim_{k\to\infty} f(\mathbf{x}_k) = f(\mathbf{x}_*)$ linearly).

We have shown in (a) that $\mathbf{x}_* = \mathbf{0}$, and in (b) that $\mathbf{x}_k = \begin{bmatrix} \left(\frac{b-a}{a+b}\right)^k b \\ \left(\frac{a-b}{a+b}\right)^k a \end{bmatrix}$. Then,

$$\begin{aligned}
\lim_{k\to\infty} \frac{\|\mathbf{x}_{k+1} - \mathbf{x}_*\|}{\|\mathbf{x}_k - \mathbf{x}_*\|} &= \sqrt{\frac{\left(\frac{b-a}{a+b}\right)^{2(k+1)} b^2 + \left(\frac{a-b}{a+b}\right)^{2(k+1)} a^2}{\left(\frac{b-a}{a+b}\right)^{2k} b^2 + \left(\frac{a-b}{a+b}\right)^{2k} a^2}} \\
&= \sqrt{\frac{\left(\frac{b-a}{a+b}\right)^{2} b^2 + \left(\frac{a-b}{a+b}\right)^{2} a^2}{b^2 + a^2}} \\
&= \sqrt{\left(\frac{a-b}{a+b}\right)^2 \frac{b^2 + a^2}{b^2 + a^2}} \\
&= \left|\frac{a-b}{a+b}\right|
\end{aligned}$$

Similarly, we know from (a) that $f(\mathbf{x}_*) = \mathbf{0}$, and from (b) that $f(\mathbf{x}_k) = \frac{ab^2+ba^2}{2}\left(\frac{b-a}{a+b}\right)^{2k}$. Therefore,

$$\begin{aligned}
\lim_{k\to\infty} \frac{|f(\mathbf{x}_{k+1}) - f(\mathbf{x}_*)|}{|f(\mathbf{x}_k) - f(\mathbf{x}_*)|} &= \frac{\left|\frac{ab^2+ba^2}{2}\left(\frac{b-a}{a+b}\right)^{2(k+1)}\right|}{\left|\frac{ab^2+ba^2}{2}\left(\frac{b-a}{a+b}\right)^{2k}\right|} \\
&= \frac{\left|\left(\frac{b-a}{a+b}\right)^{2k+2}\right|}{\left|\left(\frac{b-a}{a+b}\right)^{2k}\right|} \\
&= \left|\left(\frac{b-a}{a+b}\right)^2\right| \\
&= \left|\frac{b-a}{a+b}\right|^2
\end{aligned}$$

□

2. Implement steepest descent method and Newton method, both with backtracking line search, for minimizing a function of the form

$$f(x_1, \ldots, x_{100}) = \sum_{j=1}^{100} c_j x_j - \sum_{i=1}^{500} \log\left(b_i - \sum_{j=1}^{100} a_{ij} x_j\right).$$

Your implementation just needs to work for this specific objective function (as opposed to an arbitrary f) but it should allow for (i) arbitrary input parameters $A \in \mathbb{R}^{500\times 100}$, $\mathbf{b} \in \mathbb{R}^{500}$, and $\mathbf{c} \in \mathbb{R}^{100}$, (ii) arbitrary backtracking line search parameters $c \in (0,1)$ and $\rho \in (0,1)$, (iii) arbitrary starting point $\mathbf{x}_0$ and initial step size α_0, (iv) arbitrary tolerance $\varepsilon > 0$ for the stopping conditions (i.e., $\|\nabla f(\mathbf{x}_k)\| \le \varepsilon$ for steepest descent, $\lambda_k^2/2 \le \varepsilon$ for Newton).

We have:

$$\frac{\partial f}{\partial x_k} = c_k - \sum_{i=1}^{500} \frac{-a_{ik}}{b_i - \sum_{j=1}^{100} a_{ij}x_j} = c_k + \sum_{i=1}^{500} \frac{a_{ik}}{b_i - \sum_{j=1}^{100} a_{ij}x_j}$$

$$\frac{\partial^2 f}{\partial x_k \partial x_l} = -\sum_{i=1}^{500} \frac{-a_{ik}a_{il}}{(b_i - \sum_{j=1}^{100} a_{ij}x_j)^2} = \sum_{i=1}^{500} \frac{a_{ik}a_{il}}{(b_i - \sum_{j=1}^{100} a_{ij}x_j)^2}$$

(a) Note that this is an unconstrained optimization problem but the domain of this function is

$$\Omega := \left\{\mathbf{x} \in \mathbb{R}^{100} : b_i - \sum_{j=1}^{100} a_{ij}x_j > 0 \text{ for all } i = 1, \ldots, 500\right\}.$$

Generate A and $\mathbf{b}$ randomly in a way that ensures $\Omega \neq \varnothing$, for example

```
A = randn(500, 100); b = A*randn(100, 1) + 2*rand(500,1);
```

in Matlab/Octave/Scilab syntax). Generate $\mathbf{c}$ randomly too. Set $\alpha_0 = 1$ and generate $\mathbf{x}_0$ randomly so that $\mathbf{x}_0 \in \Omega$.

(b) Let $\mathbf{x}_*$ be the output of your implementation. Let $e_k := f(\mathbf{x}_k) - f(\mathbf{x}_*)$ be the error at the kth iteration. Plot the log of the error $\log e_k$ against k in a graph, i.e., you want to see how $\log e_k$ decreases as k increases. Why did we use a log scale? What if we instead plot the error e_k against k?

(c) Do (b) for both steepest descent and Newton methods over a range of different backtracking parameters and tolerance:

$$c = 0.01,\ 0.05,\ 0.10,\ 0.25,\ 0.50,\ 0.75,\ 0.90,$$
$$\rho = 0.05,\ 0.25,\ 0.50,\ 0.75,\ 0.95,$$
$$\varepsilon = 10^{-3},\ 10^{-5},\ 10^{-8}.$$

(d) Comment on your results[1], paying particular attention to (i) the convergence rates of steepest decent and Newton methods, (ii) how the two methods depend on on different choices of c, ρ, ε.

[1] Since your numerical experiments depend on randomly generated $A, \mathbf{b}, \mathbf{c}$, and $\mathbf{x}_0$, you should repeat them at least 10 times just to be sure that what you observed is not a fluke. However, just present one set of graphs to support your conclusions.

In [1]:
```
import numpy as np
from numpy.random import randn, rand, seed
from numpy.linalg import norm, inv, solve
import matplotlib.pyplot as plt
import math
import time
```

In [2]:
```
def f(x):
    '''
     - input: x, input vector
     - output: f(x)
    '''
    return float(c_vec.T.dot(x) - np.sum(np.log(b - A.dot(x))))

def nabla_f(x):
    '''
     - input: x, input vector
     - output: gradient of f(x)
    '''
    return np.array(c_vec) + np.sum(A / (b - A.dot(x)), 0).reshape(100,
1)

def nabla2_f(x):
    '''
     - input: x, input vector
     - output: hessian of f(x)
    '''
    return (A/((b - A.dot(x))**2)).T.dot(A)
```

In [3]:

```
def btls(pk, xk, c, rho):
    '''
    inputs:
        - pk: update vector
        - xk: vector to be updated
        - c, rho: btls parameters
    outputs:
        - alpha: best step size to use for update
    Runs the backtracking line search algorithm and returns an optimal s
tep size
    '''
    alpha = 1
    xk_plus_1 = np.array(xk + alpha*pk)
    while np.min(b-A.dot(xk_plus_1)) <= 0: # iterate until update is ins
ide of Omega
        alpha /= 2
        xk_plus_1 = np.array(xk + alpha*pk)
    while f(xk + alpha*pk) > f(xk) + c*alpha*(-pk.T).dot(pk):
        alpha *= rho # run backtracking line search to satisfy Armijo co
ndition
    return alpha

def sdbtls(x0, e=1e-3, c=.01, rho=.05, max_steps=5000):
    '''
    inputs:
        - x0: initial starting vector
        - e: tolerance level
        - c, rho: btls parameters
        - max_steps: maximum iterations allowed
    outputs:
        - xk: minimizer of f using steepest descent method with
        backtracking line search algorithm
        - fxs: list of values of f(xk) for each iteration
    Runs steepest descent method for a given x0
    '''
    fxs = []
    xk = np.array(x0)
    for _ in range(max_steps):
        pk = -nabla_f(xk) # steepest descent direction
        alpha_k = btls(pk, xk, c, rho) # run backtracking line search to
find best step size
        xk = np.array(xk + alpha_k*pk) # update x_k
        fxs.append(f(xk))
        if norm(nabla_f(xk)) < e:
            return xk, fxs

    if len(fxs) == 5000:
        print("Steepest descent algorithm failed to converge with parame
ters: \
e = {}, c = {}, rho = {}".format("%.e" %e , c, rho))
        return None, []
    else:
        return xk, fxs

def newton(x0, e=1e-3, c=.01, rho=.05, max_steps=1000):
    '''
```

```
    inputs:
        - x0: initial starting vector
        - e: tolerance level
        - c, rho: btls parameters
        - max_steps: maximum iterations allowed
    outputs:
        - xk: minimizer of f using newton method with
        backtracking line search algorithm
    Runs Newton method for a given x0, after 10 steps of steepest descen
t
    '''
    xk, fxs = sdbtls(x0, max_steps=10) # Start by running steepest desce
nt for 10 iterations
    for _ in range(max_steps):
        gr = nabla_f(xk)
        gr2 = nabla2_f(xk)
        pk = solve(gr2, -gr) # obtain inverse by solving linearly for pk
        l2 = float(gr.T.dot(-pk)) # newton decrement
        if l2/2 <= e:
            return xk, fxs
        alpha_k = btls(pk, xk, c, rho)
        xk = xk + alpha_k*pk
        fxs.append(f(xk))
    print("Newton algorithm failed to converge")
```

(a) and (b)

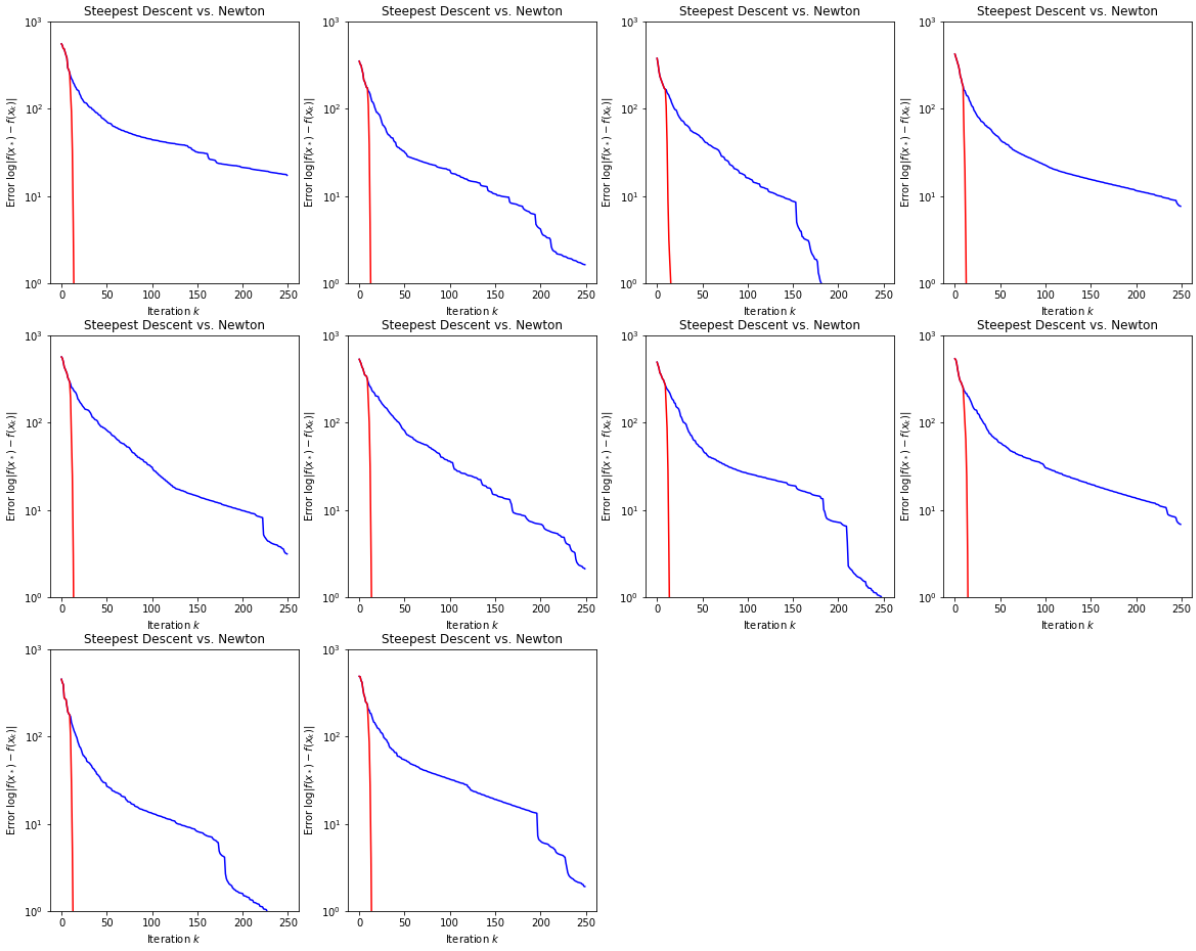
Let's run our different methods for 10 random trials to verify if our algorithms work well.

In [4]:

```
plt.figure(figsize=(20, 16))
iter_sd = []
iter_n = []
for i in range(10):
    # Generate A, x0, b, c randomly
    A = 100*randn(500, 100)
    x0 = randn(100, 1)
    b = A.dot(x0) + 200*rand(500, 1)
    c_vec = 100*randn(100, 1)

    # run steepest descent and newton algorithms
    xk_sd, fxs_sd = sdbtls(x0)
    xk_n, fxs_n = newton(x0)
    if fxs_sd: iter_sd.append(len(fxs_sd))
    iter_n.append(len(fxs_n))

    # plot log error for each
    plt.subplot(3, 4, i+1)
    if fxs_sd: plt.plot(abs(np.array(fxs_sd)[:250] - f(xk_sd)), 'b')
    plt.plot(abs(np.array(fxs_n) - f(xk_n)), 'r')
    plt.ylim(1, 1e3)
    plt.yscale('log')
    plt.minorticks_off()
    plt.xlabel(r'Iteration $k$')
    plt.ylabel(r'Error $\log|f(x_*) - f(x_k)|$')
    plt.title(r'Steepest Descent vs. Newton')
plt.show()
print("Average number of iterations for Steepest Descent: %.2f" % np.mea
n(iter_sd))
print("Average number of iterations for Newton method: %.2f" % np.mean(i
ter_n))
```




```
Average number of iterations for Steepest Descent: 1357.80
Average number of iterations for Newton method: 17.30
```

Since the Newton method converges much quicker (quadratically) than Steepest descent, we use a logarithmic scale in order to reduce significantly this skewness in iterations needed for convergence and compare both curves more easily. It allows us to see how Newton Method takes only 7 additional iterations to converge on average with a tolerance of 10^{-3} after 10 iterations of steepest descent while steepest descent alone takes around 1358 iterations to converge.

(c) and (d)

For this question, I ran each algorithm 10 times using random matrices for each parameter and averaged the numbers of iterations needed to achieve convergence. I also displayed the last run's error vs iteration plot for each parameter. If the algorithm fails to converge to $\vec{x}_*$, I did not plot the error since we need $f(\vec{x}_*)$ for this purpose.

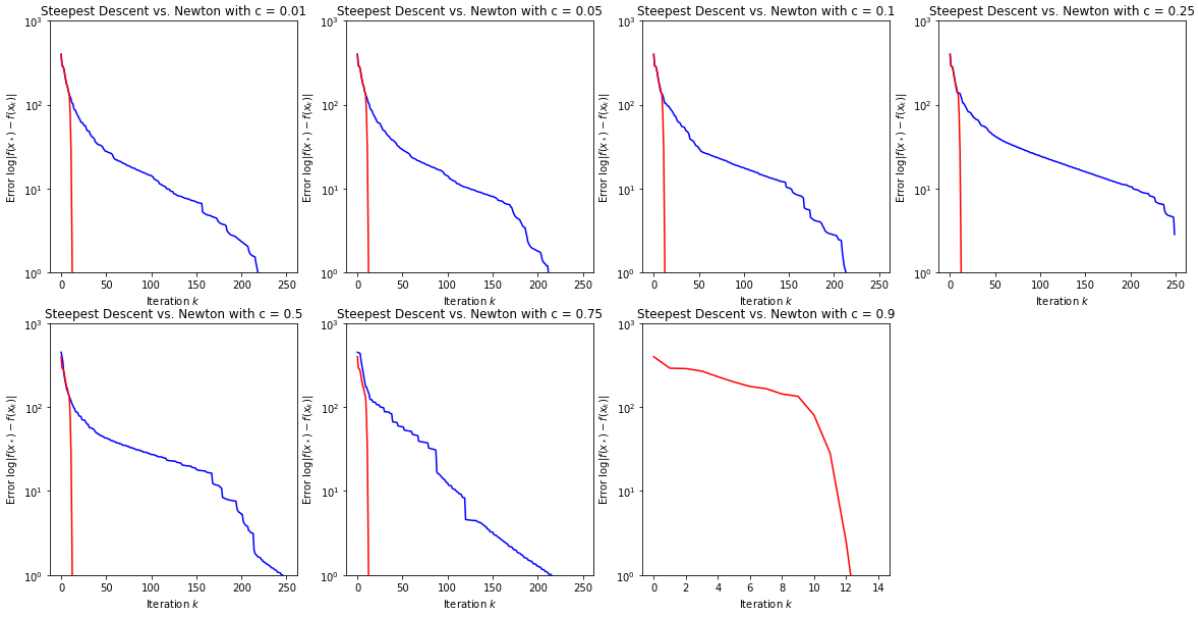
In [5]:

```
def plot_stats(sd, n, array, string):
    plt.figure(figsize=(20, 11))
    plt.subplot(231)
    plt.plot(array, np.array(sd)[:,0], 'b')
    plt.title('Average duration for Steepest Descent')
    plt.xlabel(string)
    plt.ylabel(r'Duration')
    plt.subplot(232)
    plt.plot(array, np.array(sd)[:,1], 'b')
    plt.title('Average number of Iterations for Steepest Descent')
    plt.xlabel(string)
    plt.ylabel(r'Iterations')
    plt.subplot(233)
    plt.plot(array, np.array(sd)[:,2]/10*100, 'b')
    plt.title('Percentage of convergence for Steepest Descent')
    plt.xlabel(string)
    plt.ylabel(r'%')
    plt.ylim(-9, 109)
    plt.subplot(234)
    plt.plot(array, np.array(n)[:,0], 'r')
    plt.title('Average duration for Newton')
    plt.xlabel(string)
    plt.ylabel(r'Duration')
    plt.subplot(235)
    plt.plot(array, np.array(n)[:,1], 'r')
    plt.title('Average number of Iterations for Newton')
    plt.xlabel(string)
    plt.ylabel(r'Iterations')
    plt.subplot(236)
    plt.plot(array, np.array(n)[:,2]/10*100, 'r')
    plt.title('Percentage of convergence for Newton')
    plt.xlabel(string)
    plt.ylabel(r'%')
    plt.ylim(-9, 109)
    plt.show()
```

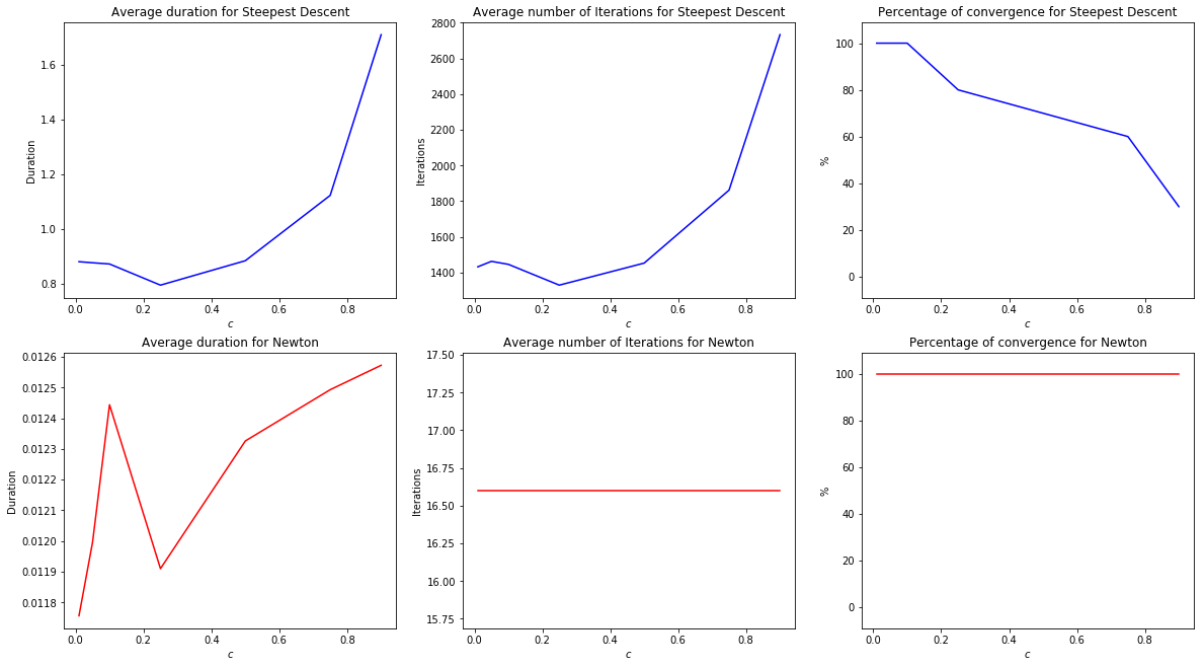
In [6]:
```
cs = [0.01, 0.05, 0.10, 0.25, 0.50, 0.75, 0.90]
plt.figure(figsize=(20, 10))
iter_sd = [[] for i in range(len(cs))]
dur_sd = [[] for i in range(len(cs))]
iter_n = [[] for i in range(len(cs))]
dur_n = [[] for i in range(len(cs))]
for k in range(10): #10 different trials
    # Generate A, x0, b, c randomly
    A = 100*randn(500, 100)
    x0 = randn(100, 1)
    b = A.dot(x0) + 200*rand(500, 1)
    c_vec = 100*randn(100, 1)
    for i in range(len(cs)):
        # run steepest descent and newton algorithms, recording time and
iterations
        tsd = time.time()
        xk_sd, fxs_sd = sdbtls(x0, c = cs[i])
        if fxs_sd:
            dur_sd[i].append(time.time()-tsd)
            iter_sd[i].append(len(fxs_sd))
        tn = time.time()
        xk_n, fxs_n = newton(x0, c = cs[i])
        dur_n[i].append(time.time()-tn)
        iter_n[i].append(len(fxs_n))

        if k == 9: # last trial
            # plot log error
            plt.subplot(2, 4, i+1)
            if fxs_sd: plt.plot(abs(np.array(fxs_sd)[:250] - f(xk_sd)),
'b')
            plt.plot(abs(np.array(fxs_n) - f(xk_n)), 'r')
            plt.ylim(1, 1e3)
            plt.yscale('log')
            plt.minorticks_off()
            plt.xlabel(r'Iteration $k$')
            plt.ylabel(r'Error $\log|f(x_*) - f(x_k)|$')
            plt.title(r'Steepest Descent vs. Newton with c = {}'.format(
cs[i]))
plt.show()
```

```
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.9, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.25, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.5, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.75, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.9, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.25, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.5, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.75, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.9, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.9, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.5, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.75, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.9, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.75, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.9, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.9, rho = 0.05
```



In [7]:
```
sd = np.array([[np.mean(dur_sd[i]), np.mean(iter_sd[i]), len(dur_sd[i])]
for i in range(len(cs))])
n = np.array([[np.mean(dur_n[i]), np.mean(iter_n[i]), len(dur_n[i])] for
i in range(len(cs))])
plot_stats(sd, n, cs, r'$c$')
```



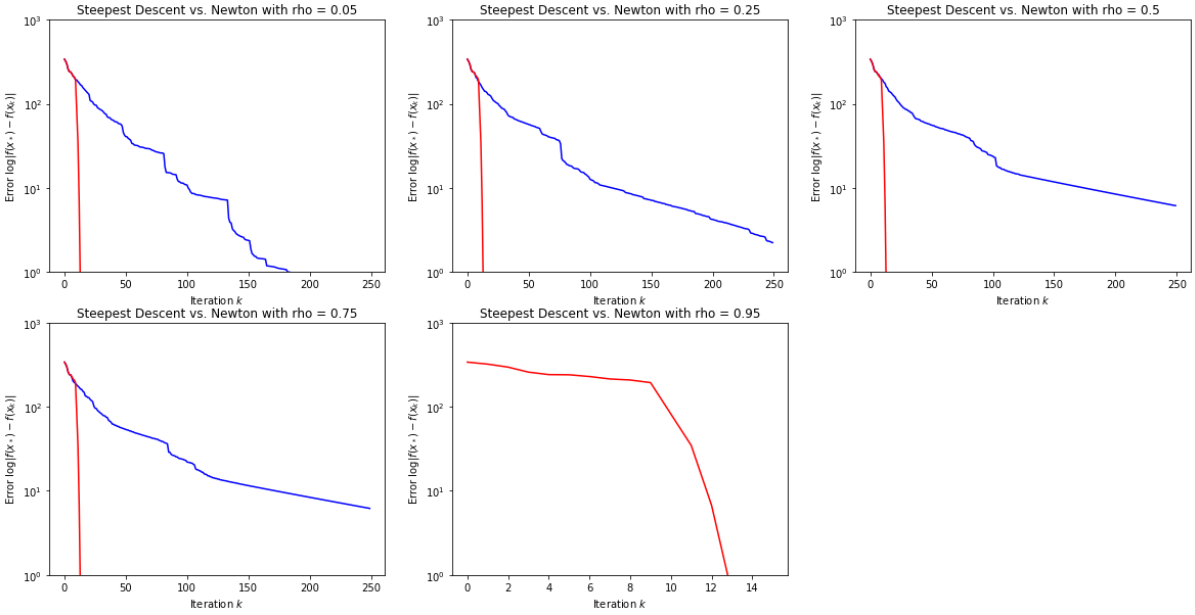
As we can see, the Steepest Descent takes more iterations and more time to converge when the backtracking parameter c increases. Additionally, Steepest Descent is a lot less likely to actually reach convergence as c increases. On the other hand, Newton method's average iterations remains the same as c increases, and the algorithm converges for every case.

In [8]:

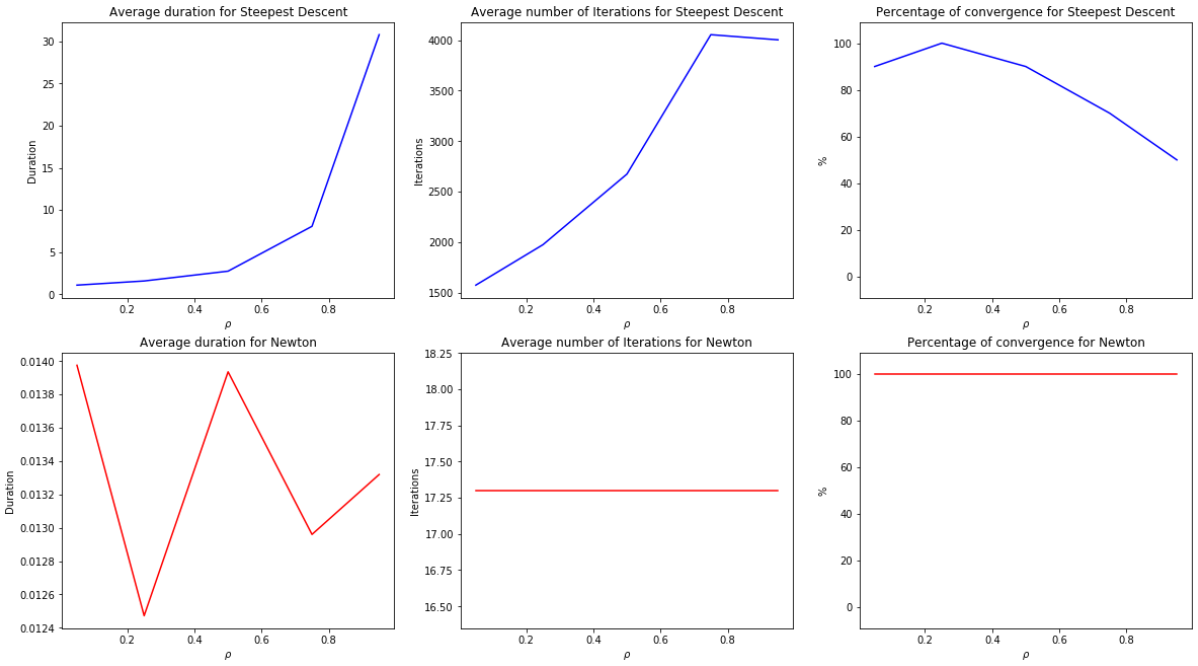
```
rhos = [0.05, 0.25, 0.50, 0.75, 0.95]
plt.figure(figsize=(20, 10))
iter_sd = [[] for i in range(len(cs))]
dur_sd = [[] for i in range(len(cs))]
iter_n = [[] for i in range(len(cs))]
dur_n = [[] for i in range(len(cs))]
for k in range(10): #10 different trials
    # Generate A, x0, b, c randomly
    A = 100*randn(500, 100)
    x0 = randn(100, 1)
    b = A.dot(x0) + 200*rand(500, 1)
    c_vec = 100*randn(100, 1)
    for i in range(len(rhos)):
        # run steepest descent and newton algorithms, recording time and
iterations
        tsd = time.time()
        xk_sd, fxs_sd = sdbtls(x0, rho = rhos[i])
        if fxs_sd:
            dur_sd[i].append(time.time()-tsd)
            iter_sd[i].append(len(fxs_sd))
        tn = time.time()
        xk_n, fxs_n = newton(x0, rho = rhos[i])
        dur_n[i].append(time.time()-tn)
        iter_n[i].append(len(fxs_n))

        if k == 9:
            # plot log error for each
            plt.subplot(2, 3, i+1)
            if fxs_sd: plt.plot(abs(np.array(fxs_sd)[:250] - f(xk_sd)),
'b')
            plt.plot(abs(np.array(fxs_n) - f(xk_n)), 'r')
            plt.ylim(1, 1e3)
            plt.yscale('log')
            plt.minorticks_off()
            plt.xlabel(r'Iteration $k$')
            plt.ylabel(r'Error $\log|f(x_*) - f(x_k)|$')
            plt.title(r'Steepest Descent vs. Newton with rho = {}'.forma
t(rhos[i]))
plt.show()
```

```
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.01, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.01, rho = 0.5
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.01, rho = 0.75
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.01, rho = 0.95
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.01, rho = 0.75
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.01, rho = 0.95
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.01, rho = 0.95
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.01, rho = 0.75
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.01, rho = 0.95
Steepest descent algorithm failed to converge with parameters: e = 1e-0
3, c = 0.01, rho = 0.95
```



In [9]:
```
sd = np.array([[np.mean(dur_sd[i]), np.mean(iter_sd[i]), len(dur_sd[i])]
for i in range(len(rhos))])
n = np.array([[np.mean(dur_n[i]), np.mean(iter_n[i]), len(dur_n[i])] for
i in range(len(rhos))])
plot_stats(sd, n, rhos, r'$\rho$')
```

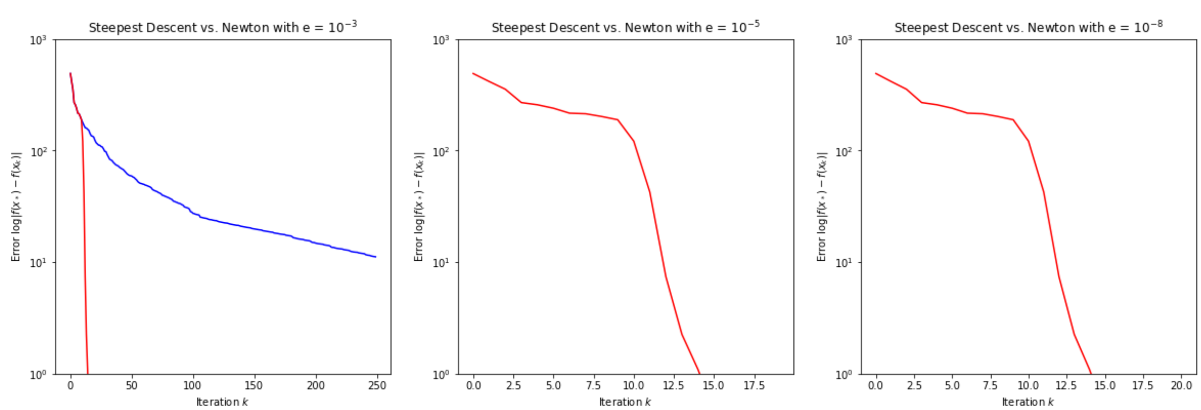


Here, I changed the backtracking parameter ρ and repeated the experiment. As we can see, the average number of iterations needed to reach convergence increase as ρ increases for Steepest Descent as well as the time required for convergence, while the iterations remain constant for Newton, and the duration is insignificant. As ρ increases, Steepest Descent is more unlikely to reach convergence, while Newton always converges.

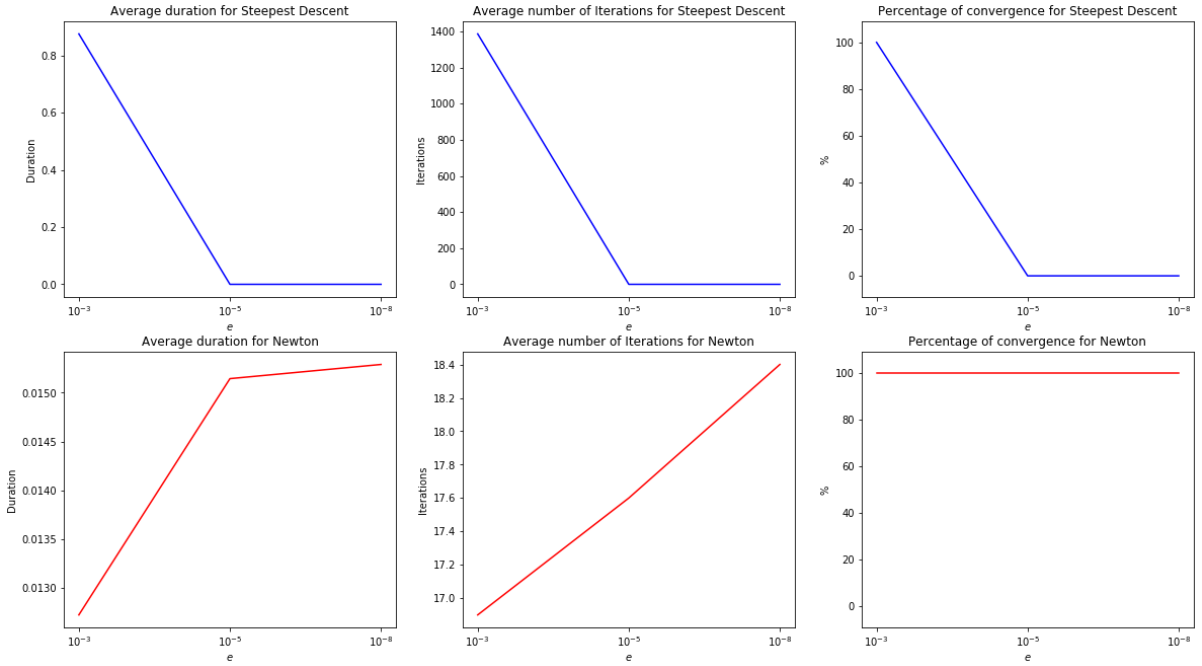
In [10]:
```
es = [1e-3, 1e-5, 1e-8]
esprint = ['$10^{-3}$', '$10^{-5}$', '$10^{-8}$']
plt.figure(figsize=(20, 6))
iter_sd = [[] for i in range(len(cs))]
dur_sd = [[] for i in range(len(cs))]
iter_n = [[] for i in range(len(cs))]
dur_n = [[] for i in range(len(cs))]
for k in range(10): #10 different trials
    # Generate A, x0, b, c randomly
    A = 100*randn(500, 100)
    x0 = randn(100, 1)
    b = A.dot(x0) + 200*rand(500, 1)
    c_vec = 100*randn(100, 1)
    for i in range(len(es)):
        # run steepest descent and newton algorithms, recording time and
iterations
        tsd = time.time()
        xk_sd, fxs_sd = sdbtls(x0, e = es[i])
        if fxs_sd:
            dur_sd[i].append(time.time()-tsd)
            iter_sd[i].append(len(fxs_sd))
        tn = time.time()
        xk_n, fxs_n = newton(x0, e = es[i])
        dur_n[i].append(time.time()-tn)
        iter_n[i].append(len(fxs_n))

        if k == 9:
            # plot log error for each
            plt.subplot(1, 3, i+1)
            if fxs_sd: plt.plot(abs(np.array(fxs_sd)[:250] - f(xk_sd)),
'b')
            plt.plot(abs(np.array(fxs_n) - f(xk_n)), 'r')
            plt.ylim(1, 1e3)
            plt.yscale('log')
            plt.minorticks_off()
            plt.xlabel(r'Iteration $k$')
            plt.ylabel(r'Error $\log|f(x_*) - f(x_k)|$')
            plt.title(r'Steepest Descent vs. Newton with e = {}'.format(
esprint[i]))
plt.show()
```

```
Steepest descent algorithm failed to converge with parameters: e = 1e-0
5, c = 0.01, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
8, c = 0.01, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
5, c = 0.01, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
8, c = 0.01, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
5, c = 0.01, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
8, c = 0.01, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
5, c = 0.01, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
8, c = 0.01, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
5, c = 0.01, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
8, c = 0.01, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
5, c = 0.01, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
8, c = 0.01, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
5, c = 0.01, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
8, c = 0.01, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
5, c = 0.01, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
8, c = 0.01, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
5, c = 0.01, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
8, c = 0.01, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
5, c = 0.01, rho = 0.05
Steepest descent algorithm failed to converge with parameters: e = 1e-0
8, c = 0.01, rho = 0.05
```



In [11]:
```
sd = np.array([[np.mean(dur_sd[0]), np.mean(iter_sd[0]), len(dur_sd[0
])], [0,0,0], [0,0,0]])
n = np.array([[np.mean(dur_n[i]), np.mean(iter_n[i]), len(dur_n[i])] for
i in range(len(es))])
plot_stats(sd, n, esprint, r'$e$')
```



As we can see, Steepest Descent is unable to converge for $e = 10^{-5}$ and $e = 10^{-8}$. However, we can see logically that Newton Method requires more steps as the tolerance e decreases.

In general, we saw that Newton method successfully converges in every case and uses only less than 20 iterations (the first 10 being steepest descent) to converge. It also takes a lot less time to converge compared to Steepest Descent in general. Although it is less expensive, Steepest Descent takes between 500 and 2000 iterations to converge on average. Additionally, Steepest Descent gets stuck at a precision level of 10^{-3} and cannot converge with more precision.

For the same matrix, it is interesting to note that the convergence rate of the Newton method does not depend on the backtracking parameters. The convergence rate remains constant. However, Steepest Descent seems to converge faster as c and ρ are smaller.

3. In the lectures, we saw that Newton method, when applied for computing square root $A^{1/2}$ of a symmetric positive definite matrix $A \in \mathbb{R}^{n\times n}$ (in fact any $n \times n$ complex matrix), yields the following iteration:

$$\begin{aligned} X_k P_k + P_k X_k &= A - X_k^2, \\ X_{k+1} &= X_k + P_k. \end{aligned}$$

(a) Using induction or otherwise, show that if $X_0 \in \mathbb{R}^{n\times n}$ commutes with A, i.e., $AX_0 = X_0 A$, then X_k commutes with A for all $k = 0, 1, 2, \ldots$. Hence deduce that the Newton iteration above may be written as

$$X_{k+1} = \frac{1}{2}(X_k + X_k^{-1} A), \tag{1}$$

assuming that X_k is invertible.

Let $F(X) = X^2 - A$. Then $[DF(X)](H) = XH + HX$.

- Base step: $AX_0 = X_0 A$ is true by our assumption.
- Induction step: Assume $X_k A_k = P_k A_k$. We want to show that $AX_{k+1} = X_{k+1} A$.
  We know that $P_k X_k + X_k P_k = A - X_k^2$. Then,

$$\begin{aligned} A - X_{k+1}^2 &= A - (X_k + P_k)^2 \\ &= A - X_k^2 - P_k X_k - X_k P_k - P_k^2 \\ &= P_k X_k + X_k P_k - P_k X_k - X_k P_k - P_k^2 \\ &= -P_k^2 \\ \Rightarrow \quad X_{k+1}(A - X_{k+1}^2) &= -X_{k+1} P_k^2 \\ \& \quad (A - X_{k+1}^2) X_{k+1} &= -P_k^2 X_{k+1} \end{aligned}$$

  Therefore, P_k commutes with X_{k+1}, we have $P_k X_k = X_k P_k \Rightarrow AX_{k+1} = X_{k+1} A$

  Now let $G_k = \frac{1}{2}(X_k^{-1} A - X_k)$. We want to show that $G_k = P_k$. Firstly, let's evaluate the derivative of $F(X)$ at G_k:

$$\begin{aligned} [DF(X_k)](G_k) &= \frac{1}{2} X_k (X_k^{-1} A - X_k) + \frac{1}{2}(X_k^{-1} A - X_k) X_k \\ &= \frac{1}{2} X_k (X_k^{-1} A - X_k) + \frac{1}{2} X_k (X_k^{-1} A - X_k) \\ &\qquad\qquad (\textit{by inductive hypothesis}) \\ &= A - X_k^2 \\ &= X_k P_k + P_k X_k \\ &= [DF(X_k)](P_k) \end{aligned}$$

  So $G_k = P_k$ and then we can deduce the following:

$$\begin{cases} P_k X_k &= \frac{1}{2}(X_k^{-1} A - X_k) X_k = \frac{1}{2}(A - X_k^2) \\ X_k P_k &= \frac{1}{2} X_k (X_k^{-1} A - X_k) = \frac{1}{2}(A - X_k^2) \end{cases},$$

  which implies that P_k commutes with X_k and therefore $AX_{k+1} = X_{k+1} A$

Finally, since $P_k = \frac{1}{2}(X_k^{-1} A - X_k)$ our newton update is the following $X_{k+1} = X_k + P_k = \frac{1}{2}(X_k + X_k^{-1} A)$. □

(b) Show that if $X \in \mathbb{R}^{n\times n}$ commutes with A, then it commutes with $A^{1/2}$.

Suppose X commutes with A, so that $XA = AX$. Since $A \in \mathbb{S}^n_{++}$ we can obtain the eigenvalue decomposition $A = V\Lambda V^{\mathrm{T}}$ where Λ is diagonal matrix with entries $\lambda_1, \ldots, \lambda_n > 0$. Therefore we have:

$$\begin{aligned} & AX = XA \\ \Rightarrow \quad & V\Lambda V^{\mathrm{T}}X = XV\Lambda V^{\mathrm{T}} \\ \Rightarrow \quad & \Lambda V^{\mathrm{T}}X = V^{\mathrm{T}}XV\Lambda V^{\mathrm{T}} \\ \Rightarrow \quad & \Lambda V^{\mathrm{T}}XV = V^{\mathrm{T}}XV\Lambda \\ \Rightarrow \quad & \lambda_i \left[V^{\mathrm{T}}XV\right]_{ij} = \left[V^{\mathrm{T}}XV\right]_{ij}\lambda_j \end{aligned}$$

Let $H = V^{\mathrm{T}}XV$. Since for any diagonal matrix D, $HD = DH$, we have either $\Lambda = 0$ or $\lambda_i = \lambda_j$, and since the eigenvalues are positive we have that $\lambda_i = \lambda_j$.
We know $A^{1/2} = V\Lambda^{1/2}V^{\mathrm{T}}$. Since H commutes with diagonal matrices, we have:

$$\begin{aligned} \Lambda^{1/2}H &= H\Lambda^{1/2} \\ \Lambda^{1/2}V^{\mathrm{T}}XV &= V^{\mathrm{T}}XV\Lambda^{1/2} \\ V\Lambda^{1/2}V^{\mathrm{T}}X &= XV\Lambda^{1/2}V^{\mathrm{T}} \\ A^{1/2}X &= XA^{1/2} \end{aligned}$$

as desired.

(c) Show that if we define error at step k by $E_k = X_k - A^{1/2}$, then

$$E_{k+1} = \frac{1}{2}X_k^{-1}E_k^2,$$

assuming that X_k is invertible and $\|\cdot\|$ is a submultiplicative matrix norm. Hence deduce that convergence is quadratic if there exists $M > 0$ such that $\|X_k^{-1}\| \leq M$ for all $k = 0, 1, 2, \ldots$.

Choose X_0 such that it commutes with A. Then, by (a), X_k commutes with A and by (b) X_k commutes with $A^{1/2}$. Additionally, by (a), $X_{k+1} = \frac{1}{2}(X_k + X_k^{-1}A)$. Therefore,

$$\begin{aligned} E_{k+1} = X_{k+1} - A^{1/2} &= \frac{1}{2}(X_k + X_k^{-1}A) - A^{1/2} \\ &= \frac{1}{2}(X_k + X_k^{-1}A - 2A^{1/2}) \\ &= \frac{1}{2}(E_k + X_k^{-1}A - A^{1/2}) \\ &= \frac{1}{2}(E_k + X_k^{-1}(A - X_kA^{1/2})) \\ &= \frac{1}{2}(E_k + X_k^{-1}(A - A^{1/2}X_k)) \\ &= \frac{1}{2}(E_k + X_k^{-1}A^{1/2}(-E_k)) \\ &= \frac{1}{2}(E_k + X_k^{-1}(X_k - E_k)(-E_k)) \qquad (\textit{Since } A^{1/2} = X_k - E_k) \end{aligned}$$

and therefore, $E_k = \frac{1}{2}(E_k - E_k + X_k^{-1}E_k^2) = \frac{1}{2}X_k^{-1}E_k^2$, as desired.
Then, we can see that:

$$\lim_{k\to\infty} \frac{\|E_{k+1}\|}{\|E_k\|} = \lim_{k\to\infty} \left\|\frac{1}{2}X_k^{-1}E_k\right\| \leq \lim_{k\to\infty} \frac{1}{2}\left\|X_k^{-1}\right\| \|E_k\| \leq \lim_{k\to\infty} \frac{1}{2}M\|E_k\|$$

Therefore, $\lim_{k\to\infty}\|E_{k+1}\| \leq \lim_{k\to\infty}\frac{1}{2}M\|E_k\|^2$ and convergence for this iteration is quadratic with rate M.

4. We will apply Newton method to obtain an analogue of (1) for computing the inverse A^{-1} of an invertible matrix $A \in \mathbb{R}^{n\times n}$ (in fact the Moore–Penrose inverse of any $m \times n$ complex matrix).

(a) Consider the function $g(X) = X^{-1}$ defined for invertible $n \times n$ matrices X. Show that the derivative of g at X is given by

$$[Dg(X)](H) = -X^{-1}HX^{-1}.$$

We want $[Dg(X)](H)$ such that:

$$\lim_{H\to 0} \frac{\|(X+H)^{-1} - X^{-1} - [Dg(X)](H)\|}{\|H\|} = 0$$

We know from lectures that:

$$\begin{aligned}
(X+H)^{-1} &= (X(I+XH))^{-1} \\
&= (I+XH)^{-1}X^{-1} \\
&= \left(I - X^{-1}H + X^{-1}HX^{-1}H - \ldots\right)X^{-1} \\
&= X^{-1} - X^{-1}HX^{-1} + X^{-1}HX^{-1}HX^{-1} - \ldots \\
&= X^{-1} - X^{-1}HX^{-1} + X^{-1}H(I - X^{-1}H + X^{-1}HX^{-1}H - \ldots)X^{-1}HX^{-1} \\
&= X^{-1} - X^{-1}HX^{-1} + X^{-1}H(I + X^{-1}H)X^{-1}HX^{-1}
\end{aligned}$$

Therefore, setting $[Dg(X)](H) = -X^{-1}HX^{-1}$, our limit becomes the following:

$$\begin{aligned}
&\lim_{H\to 0} \frac{\|X^{-1} - X^{-1}HX^{-1} + X^{-1}H(I+X^{-1}H)X^{-1}HX^{-1} - X^{-1} - X^{-1}HX^{-1}\|}{\|H\|} \\
&= \lim_{H\to 0} \frac{\|X^{-1}H(I+X^{-1}H)X^{-1}HX^{-1}\|}{\|H\|} \\
&\leq \lim_{H\to 0} \frac{\|X^{-1}\|^3\|H\|^2\|I+X^{-1}H\|}{\|H\|} \qquad \textit{(by Cauchy-Schwartz)} \\
&= \lim_{H\to 0} \|X^{-1}\|^3\|H\|\|I+X^{-1}H\| \\
&= 0
\end{aligned}$$

as desired. □

(b) Show that Newton method may be applied to an appropriate function to obtain the following iteration for computing the inverse of an invertible matrix $A \in \mathbb{R}^{n\times n}$

$$X_{k+1} = X_k(2I - AX_k). \tag{2}$$

(Hint: Emulate the univariate Newton method for computing reciprocal in Homework **1**, Problem **5**(b).) Note that like the univariate version this algorithm requires only addition and multiplication of matrices.

Let $F : \mathbb{R}^{n\times n} \to \mathbb{R}^{n\times n}, X \mapsto X^{-1} - A$.

From Newton's method for computing the inverse we know that $[DF(X)](P_k) = -F(X)$. Additionally, by (a) we know that $[DF(X)](H) = -X^{-1}HX^{-1}$ and therefore:

$$\begin{aligned}
[DF(X_k)](P_k) &= -F(X_k) \\
-X_k^{-1}P_kX_k^{-1} &= A - X_k^{-1} \\
\therefore \quad P_k &= -X_k\left(A - X_k^{-1}\right)X_k \\
&= -X_kAX_k + X_k \\
\Rightarrow \quad X_{k+1} &= X_k + P_k \\
&= X_k - X_kAX_k + X_k \\
&= 2X_k - X_kAX_k \\
&= X_k(2I - AX_k)
\end{aligned}$$

□

(c) Show that if we define error at step k by $E_k = I - AX_k$ (note that this vanishes exactly when $X_k = A^{-1}$), then

$$E_{k+1} = E_k^2 = E_{k-1}^4 = \cdots = E_0^{2^{k+1}}.$$

In other words, if (2) converges, then the convergence is quadratic. In fact one can show that for any $A \in \mathbb{R}^{m\times n}$, initializing (2) by $X_0 = \alpha A^{\mathsf{T}}$ for any $0 < \alpha < 2/\|A\|_2^2$ produces a sequence that converges to the Moore–Penrose inverse $A^{\dagger}$. In particular if $A \in \mathbb{R}^{n\times n}$ is invertible, then $\lim_{k\to\infty} X_k = A^{-1}$.

We have:

$$\begin{aligned}
X_{k+1} &= X_k(2I - AX_k) \\
AX_{k+1} &= AX_k(2I - AX_k) \\
\therefore \quad E_{k+1} &= I - AX_{k+1} \\
&= I - AX_k(2I - AX_k) \\
&= I - 2AX_k + (AX_k)^2 \\
&= (I - AX_k)^2 \\
&= E_k^2
\end{aligned}$$

(d) Implement the algorithm in (b) with the initialization described in (c) with a simple stopping criteria (e.g. stop when $\|X_{k+1} - X_k\|_F$ or $\|E_k\|_F$ is small).

i. Compare the result X_* obtained by your implementation for 2×2 matrices A with random integer entries and for a 10×10 diagonal matrices A with random rational entries with the actual A^{-1}, which you know analytically. Check the accuracy of your implementation by observing the values of $\|X_* - A^{-1}\|_F$ (this is called the *forward error*, note that you can compute this only if you already know A^{-1}).

ii. Compare the result X_* obtained by your implementation for randomly generated $n \times n$ matrices A with the result Y_* obtained by calling the matrix inversion function

of the software you use. Do this for $n = 10, 10^2, 10^3$. Check the accuracy of your implementation by comparing the values of $\|I - AX_*\|_F$ and $\|I - AY_*\|_F$ (this is called the *backward error*, note that you can compute this even if you do not know A^{-1}).

In [1]:
```
import numpy as np
from numpy.random import randn, rand, randint, seed, uniform
from numpy.linalg import norm, inv, solve, pinv
import matplotlib.pyplot as plt
import math
import time
```

In [2]:
```
def newton_inverse(A, e=1e-8, max_steps=1000):
    '''
    inputs:
        - A: initial matrix
    outputs:
        - xk: minimizer of f using newton method
    Runs Newton method for matrix inverse (without steepest descent iter
ations,
    since we already have a good starting point).
    '''
    alpha = uniform(0, 1/norm(A, ord=2)**2)
    X0 = alpha * A.T # initialize x0 as shown in part (c)

    for k in range(max_steps):
        Xk = X0.dot(2 * np.identity(A.shape[0]) - A.dot(X0))
        if norm(Xk - X0) < e:
            return Xk, k
        X0 = np.array(Xk)
    print("Newton algorithm failed to converge")
```

In [3]:
```
def backward_error(A, X):
    try:
        Y = inv(A)
    except:
        Y = pinv(A)

    I = np.identity(len(A))
    ex = norm(I-A@X)
    ey = norm(I-A@Y)

    return (ex, ey)
```

i.

Let's calculate the forward errors for 2×2 integer matrices and 10×10 diagonal matrices of rational numbers. I used a tolerance of 10^{-8}.

In [4]:
```
for _ in range(10):
    A = randint(-5000, 5000, size=(2, 2))
    Xk, k = newton_inverse(A)
    a, b, c, d = A[0, 0], A[0, 1], A[1, 0], A[1, 1]
    detA = (a*d-b*c)
    if detA:
        A_minus_1 = 1/detA*np.array([[d, -b], [-c, a]]) #inverse of a 2x
2 matrix
        print(f"Forward error obtained after {k} steps: {norm(Xk - A_min
us_1)}")
    else:
        print("A is singular")
```

```
Forward error obtained after 4 steps: 7.265469234097228e-19
Forward error obtained after 8 steps: 1.4844286914218873e-15
Forward error obtained after 7 steps: 3.408294302622768e-16
Forward error obtained after 9 steps: 9.373686084755107e-16
Forward error obtained after 5 steps: 1.2428310280723025e-17
Forward error obtained after 9 steps: 1.5739712730627153e-17
Forward error obtained after 14 steps: 1.2361806729070689e-18
Forward error obtained after 22 steps: 1.7355703022298208e-17
Forward error obtained after 9 steps: 2.737476298800061e-19
Forward error obtained after 9 steps: 4.846769225765442e-15
```

In [5]:
```
for _ in range(10):
    seed()
    A = np.diag(randint(-5000, 5000, size=10)/randint(-5000, 5000, size=
10))
    Xk, k = newton_inverse(A)
    A_minus_1 = np.diag([1/a for a in np.diag(A)]) # inverse of each ent
ry
    print(f"Forward error obtained after {k} steps: {norm(Xk - A_minus_
1)}")
```

```
Forward error obtained after 19 steps: 2.2898349882893854e-16
Forward error obtained after 13 steps: 1.1102230246251565e-16
Forward error obtained after 14 steps: 6.38529066861312e-16
Forward error obtained after 14 steps: 4.47545209131181e-16
Forward error obtained after 17 steps: 4.577566798522237e-16
Forward error obtained after 15 steps: 1.8343894894033213e-15
Forward error obtained after 14 steps: 3.3335590258932494e-16
Forward error obtained after 12 steps: 5.551115123125783e-17
Forward error obtained after 16 steps: 3.5579140431534355e-15
Forward error obtained after 18 steps: 1.3334236103572998e-15
```

ii.

In [6]:

```
rate = 0
ratet = 0
for n in [10, 100, 1000]:
    print("testing matrix: ", (n, n))
    I = np.identity(n)
    for _ in range(10):
        A = randn(n, n)
        Xk, k = newton_inverse(A)
        tn = time.time()
        be_newton = norm(I-A.dot(Xk))
        tn = time.time() - tn
        print("Backward error obtained after {} steps (duration {:.2e}):
{:.2e}"\
              .format(k, tn, be_newton))
        tnp = time.time()
        be_numpy = norm(I-A.dot(inv(A)))
        tnp = time.time() - tnp
        print("Backward error of numpy.linalg.inv (duration {:.2e}): {:.
2e}"\
             .format(tnp, be_numpy))
        if tn - tnp < 0:
            print("shorter duration for newton method")
            ratet+=1
        else:
            print("numpy is shorter")
        if be_newton - be_numpy < 0:
            print("better error for newton method!\n")
            rate+=1
        else:
            print("numpy wins\n")
```

```
testing matrix:  (10, 10)
Backward error obtained after 19 steps (duration 1.69e-05): 6.45e-15
Backward error of numpy.linalg.inv (duration 1.85e-04): 8.34e-15
shorter duration for newton method
better error for newton method!

Backward error obtained after 14 steps (duration 2.60e-05): 1.62e-15
Backward error of numpy.linalg.inv (duration 1.02e-04): 2.64e-15
shorter duration for newton method
better error for newton method!

Backward error obtained after 17 steps (duration 2.31e-05): 3.51e-15
Backward error of numpy.linalg.inv (duration 7.89e-05): 3.83e-15
shorter duration for newton method
better error for newton method!

Backward error obtained after 17 steps (duration 2.10e-05): 4.77e-15
Backward error of numpy.linalg.inv (duration 7.82e-05): 4.44e-15
shorter duration for newton method
numpy wins

Backward error obtained after 16 steps (duration 2.69e-05): 1.42e-15
Backward error of numpy.linalg.inv (duration 8.30e-05): 1.37e-15
shorter duration for newton method
numpy wins

Backward error obtained after 14 steps (duration 2.22e-05): 1.26e-15
Backward error of numpy.linalg.inv (duration 1.57e-04): 1.49e-15
shorter duration for newton method
better error for newton method!

Backward error obtained after 17 steps (duration 2.69e-05): 5.04e-15
Backward error of numpy.linalg.inv (duration 1.07e-04): 4.90e-15
shorter duration for newton method
numpy wins

Backward error obtained after 18 steps (duration 2.19e-05): 3.33e-15
Backward error of numpy.linalg.inv (duration 7.18e-05): 3.40e-15
shorter duration for newton method
better error for newton method!

Backward error obtained after 16 steps (duration 1.60e-05): 3.57e-15
Backward error of numpy.linalg.inv (duration 8.20e-05): 2.78e-15
shorter duration for newton method
numpy wins

Backward error obtained after 15 steps (duration 2.10e-05): 2.23e-15
Backward error of numpy.linalg.inv (duration 9.99e-05): 2.20e-15
shorter duration for newton method
numpy wins

testing matrix:  (100, 100)
Backward error obtained after 22 steps (duration 7.22e-05): 6.14e-14
Backward error of numpy.linalg.inv (duration 6.19e-04): 1.83e-13
shorter duration for newton method
better error for newton method!
```

```
Backward error obtained after 20 steps (duration 6.01e-05): 5.36e-14
Backward error of numpy.linalg.inv (duration 4.95e-04): 1.41e-13
shorter duration for newton method
better error for newton method!

Backward error obtained after 30 steps (duration 5.70e-05): 9.43e-13
Backward error of numpy.linalg.inv (duration 5.48e-04): 2.15e-12
shorter duration for newton method
better error for newton method!

Backward error obtained after 22 steps (duration 8.77e-05): 4.05e-14
Backward error of numpy.linalg.inv (duration 6.38e-04): 1.12e-13
shorter duration for newton method
better error for newton method!

Backward error obtained after 24 steps (duration 5.79e-05): 1.46e-13
Backward error of numpy.linalg.inv (duration 4.79e-04): 3.80e-13
shorter duration for newton method
better error for newton method!

Backward error obtained after 25 steps (duration 5.70e-05): 2.18e-13
Backward error of numpy.linalg.inv (duration 4.16e-04): 5.98e-13
shorter duration for newton method
better error for newton method!

Backward error obtained after 30 steps (duration 6.29e-05): 1.06e-13
Backward error of numpy.linalg.inv (duration 4.82e-04): 2.77e-13
shorter duration for newton method
better error for newton method!

Backward error obtained after 33 steps (duration 6.70e-05): 2.15e-13
Backward error of numpy.linalg.inv (duration 4.88e-04): 6.20e-13
shorter duration for newton method
better error for newton method!

Backward error obtained after 24 steps (duration 5.39e-05): 1.20e-13
Backward error of numpy.linalg.inv (duration 4.26e-04): 3.47e-13
shorter duration for newton method
better error for newton method!

Backward error obtained after 22 steps (duration 5.72e-05): 4.47e-14
Backward error of numpy.linalg.inv (duration 4.94e-04): 1.15e-13
shorter duration for newton method
better error for newton method!

testing matrix:  (1000, 1000)
Backward error obtained after 28 steps (duration 2.11e-02): 1.18e-12
Backward error of numpy.linalg.inv (duration 6.60e-02): 2.28e-11
shorter duration for newton method
better error for newton method!

Backward error obtained after 28 steps (duration 1.93e-02): 1.31e-12
Backward error of numpy.linalg.inv (duration 6.52e-02): 2.36e-11
shorter duration for newton method
better error for newton method!

Backward error obtained after 27 steps (duration 2.67e-02): 9.85e-13
```

```
Backward error of numpy.linalg.inv (duration 9.31e-02): 1.90e-11
shorter duration for newton method
better error for newton method!

Backward error obtained after 26 steps (duration 2.13e-02): 7.56e-13
Backward error of numpy.linalg.inv (duration 6.61e-02): 1.57e-11
shorter duration for newton method
better error for newton method!

Backward error obtained after 31 steps (duration 2.09e-02): 1.13e-12
Backward error of numpy.linalg.inv (duration 6.47e-02): 2.27e-11
shorter duration for newton method
better error for newton method!

Backward error obtained after 34 steps (duration 1.93e-02): 2.17e-12
Backward error of numpy.linalg.inv (duration 5.96e-02): 3.83e-11
shorter duration for newton method
better error for newton method!

Backward error obtained after 26 steps (duration 2.30e-02): 7.05e-13
Backward error of numpy.linalg.inv (duration 6.65e-02): 1.38e-11
shorter duration for newton method
better error for newton method!

Backward error obtained after 27 steps (duration 2.21e-02): 1.32e-12
Backward error of numpy.linalg.inv (duration 6.55e-02): 2.65e-11
shorter duration for newton method
better error for newton method!

Backward error obtained after 31 steps (duration 2.28e-02): 1.73e-12
Backward error of numpy.linalg.inv (duration 9.10e-02): 3.34e-11
shorter duration for newton method
better error for newton method!

Backward error obtained after 35 steps (duration 2.83e-02): 9.47e-12
Backward error of numpy.linalg.inv (duration 7.48e-02): 1.91e-10
shorter duration for newton method
better error for newton method!
```

In [7]:
```
print("Newton wins error {:.2f}% of the time".format(rate/30*100))
print("Newton wins duration {:.2f}% of the time".format(ratet/30*100))
```

```
Newton wins error 83.33% of the time
Newton wins duration 100.00% of the time
```

We can see that out Newton implementation with tolerance of 10^{-8} does better than NumPy's inversion function more than 80% of the time. It is therefore incredibly precise, and for all our bigger matrices ($100 \times 100, 1000 \times 1000$), our implementation has a smaller error rates than NumPy.

Additionally, our implementation also takes less time to compute the inverse than NumPy does, for all matrices.