Large Scale Data Analysis ELEC-E5431
Name: Nguyen Xuan Binh
Student ID: 887799

Problem Setup:

The optimization problem to be addressed is a simple quadratic function minimization, that is,

$$\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x}$$

where the matrix $\mathbf{A}$ and vector $\mathbf{b}$ are appropriately generated. Use the same $\mathbf{A}$ and $\mathbf{b}$ while comparing different methods.

Hints:

- $\mathbf{A}$ should be such that $\mathbf{x}^T\mathbf{A}\mathbf{x}$ is non-negative, that is, $\mathbf{A}$ is positive semi-definite, and $\mathbf{b}$ should be in the range of $\mathbf{A}$.

- In fact, by solving the above unconstrained minimization problem, you solve a system of linear equations $\mathbf{A}\mathbf{x} = \mathbf{b}$. Indeed, the gradient of the objective function is $\mathbf{A}\mathbf{x} - \mathbf{b}$, and it should be equal to 0 at optimality. Thus, you can find optimal $\mathbf{x}^*$ using back-slash (or matrix inversion followed by computing the product $\mathbf{A}^{-1}\mathbf{b}$) operators in MATLAB. The optimal objective value can be then obtained by simply substituting such $\mathbf{x}$ into the objective of the above optimization problem. It is suitable for small and mid size problems, but the matrix inversion is prohibitively too expensive to be able to solve a system of linear equations for large scale problems. Thus, the only option for large scale problems is the use of algorithms that you implement in this assignment!

To be able to produce convergence figures for the algorithms that you test, let the dimension of $\mathbf{x}$ be 100 variables or few 100's (but after producing the figures also play with higher dimensions to see when the matrix inversion fails, but the large scale optimization methods still work fine and some also quite fast).

Set the tolerance parameter for the stopping criterion for checking the convergence to 10^{-5}. For example, check if $\|\nabla f(\mathbf{x})\| \leq 10^{-5}$, and limit the total number of iterations by 5000 if the predefined tolerance is still not achieved.

Task 1: *Gradient Descent Algorithm.*

Implement Gradient Descent Algorithm for solving the above optimization problem. Use correctly selected fixed step size α. Draw the experimental convergence rate, i.e., draw the plot $\log \|\mathbf{x}_k - \mathbf{x}^*\|_2$ versus iteration number k, for the algorithm and compare it with the theoretically predicted one.

2.4 Steepest Descent (Gradient Descent)

The iterate is given as

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k), \quad \alpha_k > 0.$$

How to select the step size α_k:

1. Fixed: use rules based on L and μ (trivial),
2. Backtracking (computationally easy),
3. exact line search (computationally may be hard).

For the above ways of step size selection 2 and 3, we typically have global convergence at unspecified rate.

The "greedy" strategy of getting good decrease in the current search direction may lead to better practical convergence results.

For the above way of step size selection, fixed step size selection focuses on convergence rate.

Task 2: *Conjugate Gradient Algorithm.*

Implement Conjugate Gradient Algorithm for solving the above optimization problem. Use correctly selected fixed step size. Draw the experimental convergence rate, i.e., draw the plot $\log \|\mathbf{x}_k - \mathbf{x}^*\|_2$ versus iteration number k, for the algorithm and compare it with the theoretically predicted one.

2.6 Conjugate Gradient (CG)

The iterate is given as

$$x_{k+1} = x_k + \alpha_k \rho_k, \rho_k = -\nabla f(x_k) + \delta_k \rho_{k-1}$$

The same as heavy-ball with $\beta_k = \frac{\alpha_k \delta_k}{\alpha_{k-1}}$, but in CG α_k and β_k are selected in particular way and the method does it itself.

CG can be implemented in a way that does not require knowledge (estimate) of L and μ:

- Choose α_k to minimize f along ρ_k,
- Choose δ_k by a variety of formulae (Fletcher-Reeves, Polak-Ribiere, etc.) all of these formulae are equivalent if f is convex quadratic, e.g.,

$$\delta_k = \frac{\|\nabla f(x_k)\|_2^2}{\|\nabla f(x_{k-1})\|_2^2}.$$

Restarting periodically with $\rho_k = -\nabla f(x_k)$ is useful, e.g., every n iterations or when ρ_k is not a descent direction.

For quadratic f: convergence analysis is based on eigenvalues of A and Chebyshev polynomials (min-max argument), linear convergence with rate $1 - \frac{2}{\sqrt{\varkappa}}$ (like heavy-ball).

Task 3: *FISTA.*

Implement FISTA for solving the above optimization problem. Use correctly selected fixed step size. Draw the experimental convergence rate, i.e., draw the plot $\log \|\mathbf{x}_k - \mathbf{x}^*\|_2$ versus iteration number k, for the algorithm and compare it with the theoretically predicted one.

2.8 FISTA (Beck & Teboulle 2009)

Simpler generic convergence analysis compared to Nesterov, adopted to composite objective function - proximal method. Otherwise the accelaration idea is the same by Nesterov.

Algorithm:

Initialize: Choose x_0; set $y_1 = x_0, t_1 = 1$.

Iterate:

$$x_k = y_k - \frac{1}{L}\nabla f(y_k)$$
$$t_{k+1} = \frac{1}{2}\left(1 + \sqrt{1 + 4t_k^2}\right)$$
$$y_{k+1} = x_k + \frac{t_k - 1}{t_{k+1}}(x_k - x_{k-1}).$$

For both strongly and weakly convex f, converges with $\frac{1}{k^2}$.

When L is not known, increase an estimate of L until it's big enough.

Task 4: *Coordinate Descent.*

Implement the Coordinate Descent method for solving the above optimization problem. Use correctly selected fixed step size. Draw the experimental convergence rate, i.e., draw the plot $\log \|\mathbf{x}_k - \mathbf{x}^*\|_2$ versus iteration number k, for the algorithm.

4.4.1 Deterministic and Stochastic CD

The update rule is

$$x_{j+1,i_j} = x_{j,i_j} - \alpha_j[\nabla f(x_j)]_{i_j}.$$

- Deterministic: choose i_j in fixed order (cyclic).
- Stochastic: choose i_j at random.

Convergence: Deterministic (Luo & Tseng 1992) – Linear rate (Beck & Tetruashvili, 2013).

Stochastic – linear rate (Nesterov, 2012).

Task 5: *Comparisons.*

Compare the results (in terms of the iterations required and the overall computation time) for different methods (including, for example, the standard MATLAB back-slash operator) and draw your overall conclusions. Observe up to which dimension Python or MATLAB still can invert a matrix, that is, define the dimension after which the problem turns to be large scale in the context of your implementation.

Because my laptop can still invert a matrix of dimension of 10000 pretty fast (20-30 seconds), and the matrix is already very heavy. If I increase the dimension even higher, my laptop will not have enough memory to store the matrix, so I am not sure what is the boundary between the large and small scale in my case. Therefore, I choose dimension 10 as small, 100 as large and 1000 as huge scale, and see how the algorithms perform for each case.

You can see all of the tasks completed below in the attached PDF file generated from the ipynb file. Additionally, you can run the file optimization.ipynb in the zipped project file. This file contains every information, from generating matrix data, algorithm implementations to convergence rate analysis.

optimization

January 18, 2023

1 Importing the libraries

[35]:
```
# Importing libraries
import import_ipynb
import numpy as np
import time
import math
import copy
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure
from matplotlib import ticker
```

2 Generate the positive definite matrix A, the vector b in the range of A and the optimal solution x*

[36]:
```
# Method of generating a positive semidefinite matrix
# 1. Generate a random square matrix
# 2. multiply it by its own transposition
# 3. we have obtained a positive semi-definite matrix.
def generateRandomPositiveSemidefiniteMatrix(size, scaleDown):
    randomMatrix = np.random.rand(size, size)
    positive_semidefinite_matrix = np.matmul(randomMatrix, randomMatrix.
↪transpose()) / scaleDown
    return positive_semidefinite_matrix

# Generate linearly spacing vector from 1 to size for x*
def generateLinearOptimalX(size):
    return np.arange(1, size + 1, 1).astype(int)

# A matrix is positive semidefinite if all of its eigenvalues are nonnegative
# Note: this matrix should be symmetric
def isPositiveSemidefinite(A):
    return np.all(np.linalg.eigvals(A) >= 0)

# A matrix does not have an inverse if its determinant is equal to 0
def inverse(A):
```

```
    return np.linalg.inv(A)
```

3 There are three different test cases: the small scale with dimension of 10, the large scale with dimension of 100 and the huge scale with the dimension of 1000

[50]:
```
scales = ["small", "large", "huge"]

for scale in scales:

    if scale == "small":
        size = 10
        scaleDown = 1
    elif scale == "large":
        size = 100
        scaleDown = 10
    elif scale == "huge":
        size = 1000
        scaleDown = 100
    print(f"Data generation for the {scale} scale test. Dimension: {size}")

    start = time.time()
    A = generateRandomPositiveSemidefiniteMatrix(size, scaleDown)
    end = time.time()

    print("The matrix A is")
    print(A)

    print("\nTime required to generate the matrix A is")
    print(f"{end - start} seconds")

    # print(f"Is the matrix A positive semidefinite?:␣
 ↪{isPositiveSemidefinite(A)}")

    start = time.time()
    invA = inverse(A)
    end = time.time()

    print("\nThe inverse of matrix A is")
    print(invA)

    print("\nTime required to invert the matrix A is")
    print(f"{end - start} seconds")
```

```python
    # The optimal solution is designed as the natural numbers: 1,2,3,4,5 and so␣
↪on
    x_opt = generateLinearOptimalX(size)
    print("\nThe optimal solution x* is")
    print(x_opt)

    # b is in the range of A
    # In other words, b is a linear combination of the columns of matrix A
    b = A.dot(x_opt)
    print("\nThe vector b is")
    print(b)
    print("\n\n\n")

    # So we have the identities:
    # Ax* = b or Ax - b = 0
    # A^-1b = x*

    np.save(f"data/{scale}Matrix.npy", A)
    np.save(f"data/{scale}Vector.npy", b)
    np.save(f"data/{scale}Solution.npy", x_opt)
```

```
Data generation for the small scale test. Dimension: 10
The matrix A is
[[5.18739286 3.82006574 3.02467891 4.67019152 4.4263327  3.85433794
  3.29062272 3.21637545 3.68674808 2.44198635]
 [3.82006574 3.95598735 2.96634602 3.74129605 3.34534356 2.84943095
  2.56262218 2.73734196 3.29195817 2.18187658]
 [3.02467891 2.96634602 2.83309499 2.97713416 2.88903791 2.4030659
  2.41896517 2.51807188 2.29166691 1.45231162]
 [4.67019152 3.74129605 2.97713416 4.79083468 4.09810121 3.76307602
  3.28039034 3.1383191  3.33488093 2.4079852 ]
 [4.4263327  3.34534356 2.88903791 4.09810121 4.58298378 3.45154963
  2.71213933 3.02208139 3.06182015 1.88265053]
 [3.85433794 2.84943095 2.4030659  3.76307602 3.45154963 3.25389305
  2.68453657 2.45483449 2.70832725 1.83173519]
 [3.29062272 2.56262218 2.41896517 3.28039034 2.71213933 2.68453657
  2.93555239 2.44009243 2.22225287 1.65173892]
 [3.21637545 2.73734196 2.51807188 3.1383191  3.02208139 2.45483449
  2.44009243 2.54214198 2.1846371  1.47177919]
 [3.68674808 3.29195817 2.29166691 3.33488093 3.06182015 2.70832725
  2.22225287 2.1846371  3.49206426 1.76574634]
 [2.44198635 2.18187658 1.45231162 2.4079852  1.88265053 1.83173519
  1.65173892 1.47177919 1.76574634 1.97690738]]

Time required to generate the matrix A is
0.0010020732879638672 seconds
```

```
The inverse of matrix A is
[[  6.31369782   2.03077804   3.30784445   2.05512423   0.36795623
   -4.62446685   1.55283315  -8.91786292  -3.60674967  -2.47592768]
 [  2.03077804   9.03651895  -3.97172084  -1.46664953   1.98667402
   -0.30673841   4.40954976  -6.20830477  -5.1248167   -3.87031225]
 [  3.30784445  -3.97172084   9.37011006   5.16454873   0.70198202
   -5.92456378  -0.3777639  -10.04340755  -0.35261707   0.05186895]
 [  2.05512423  -1.46664953   5.16454873   7.43147417   1.52950198
   -7.38450401   1.12373609  -9.37834034  -1.00926741  -1.43566414]
 [  0.36795623   1.98667402   0.70198202   1.52950198   4.06457578
   -4.36667293   3.85996682  -7.56401338  -1.54658383  -1.0630487 ]
 [ -4.62446685  -0.30673841  -5.92456378  -7.38450401  -4.36667293
   12.91607466  -5.0174079   15.37804544   2.37932681   2.20718637]
 [  1.55283315   4.40954976  -0.3777639    1.12373609   3.85996682
   -5.0174079    6.74968797 -10.03170498  -2.86005041  -2.51956657]
 [ -8.91786292  -6.20830477 -10.04340755  -9.37834034  -7.56401338
   15.37804544 -10.03170498  33.22340482   8.04006146   6.09008591]
 [ -3.60674967  -5.1248167   -0.35261707  -1.00926741  -1.54658383
    2.37932681  -2.86005041   8.04006146   5.00478556   2.80176085]
 [ -2.47592768  -3.87031225   0.05186895  -1.43566414  -1.0630487
    2.20718637  -2.51956657   6.09008591   2.80176085   3.58237677]]

Time required to invert the matrix A is
0.0009872913360595703 seconds

The optimal solution x* is
[ 1  2  3  4  5  6  7  8  9 10]

The vector b is
[192.20597715 160.7030465  130.45422714 185.47955251 169.34526813
 149.71853723 135.04951123 130.43527391 144.16316175 100.19547579]




Data generation for the large scale test. Dimension: 100
The matrix A is
[[3.13994183 2.488769   2.61732092 … 2.32355668 2.47360974 2.40710816]
 [2.488769   3.86331209 2.75016916 … 2.51931198 2.72744451 2.82935441]
 [2.61732092 2.75016916 3.45267036 … 2.5659738  2.60370428 2.51052948]
 …
 [2.32355668 2.51931198 2.5659738  … 3.14748759 2.35775558 2.41655926]
 [2.47360974 2.72744451 2.60370428 … 2.35775558 3.35645681 2.5015893 ]
 [2.40710816 2.82935441 2.51052948 … 2.41655926 2.5015893  3.44387547]]

Time required to generate the matrix A is
0.0 seconds
```

```
The inverse of matrix A is
[[ 18442.25324954 -13676.98079602 -24967.96961572 …   9012.18119031
  -10628.70876097  -5349.44837691]
 [-13676.98079597  10330.50762925  18635.67831259 …  -6682.64986148
    8059.02048141   3930.35998728]
 [-24967.96961574  18635.67831267  33951.48959648 … -12237.03885599
   14465.66155191   7213.92584786]
 …
 [  9012.18119033  -6682.64986152 -12237.03885602 …   4434.71391424
   -5162.65309222  -2605.73539014]
 [-10628.70876088   8059.02048137  14465.66155178 …  -5162.65309216
    6369.35043999   3059.75612469]
 [ -5349.44837692   3930.3599873    7213.92584787 …  -2605.73539014
    3059.75612472   1577.84389255]]

Time required to invert the matrix A is
0.0039980411529541016 seconds

The optimal solution x* is
[  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
  91  92  93  94  95  96  97  98  99 100]

The vector b is
[11951.4115316  13668.11060914 13128.3439263  13446.41130468
 11396.74358665 11536.42040716 11988.35414693 12879.58711637
 11583.57842884 12845.35112076 11614.20913591 13216.49209787
 13229.22664941 13513.64521322 11989.24061182 12044.72498256
 12472.99432521 12577.17238309 11386.44514139 13589.6475785
 13342.42551611 12432.27946437 13539.67697297 14194.83510356
 13067.33397387 12472.70724337 13109.32227699 10339.41769487
 12179.19629227 14106.66874556 12536.63839371 12443.16129939
 12666.12477877 11941.17426665 13158.52635285 13153.74186299
 12730.74100369 12515.77518577 12487.5578692  12614.73106332
 11893.75445949 13837.71474911 12020.31168545 11480.49611585
 11769.8904268  12605.2935294  12347.12761509 13052.13205585
 13309.06504383 12644.22308362 12631.98078663 12961.90355795
 13034.46437815 12244.63325669 13129.16772386 11740.37817754
 12774.00697391 13193.42876032 11782.56774397 12176.45877321
 12688.60833572 12503.98770786 13188.4320737  12397.85396559
 13301.72422593 12280.79083091 11602.52164541 12624.5807516
 13842.09258105 12080.52747419 13046.84671288 14717.04320886
 12694.69158636 12399.5084073  13335.03107025 11627.10817166
 12600.18097037 11275.94272285 13181.0396423  13639.14539868
 12757.78370794 13104.98871563 12445.35888241 13974.87166027
```

```
 11060.44962511 12371.85740991 11157.31066671 11766.00586107
 12787.45699238 13889.46329881 13396.38475157 11996.09469735
 13505.67298862 11887.65152997 11545.07427836 11605.48391919
 13260.68733659 12215.80212647 12881.82173727 12650.63877369]
```

```
Data generation for the huge scale test. Dimension: 1000
The matrix A is
[[3.46670326 2.4664962  2.57697622 … 2.60897989 2.48262417 2.52327387]
 [2.4664962  3.26569944 2.42777349 … 2.49552908 2.39568779 2.40236126]
 [2.57697622 2.42777349 3.38609734 … 2.5507125  2.48939967 2.43790291]
 …
 [2.60897989 2.49552908 2.5507125  … 3.42260011 2.48801867 2.45307578]
 [2.48262417 2.39568779 2.48939967 … 2.48801867 3.16346634 2.38766003]
 [2.52327387 2.40236126 2.43790291 … 2.45307578 2.38766003 3.1789813 ]]

Time required to generate the matrix A is
0.02899932861328125 seconds

The inverse of matrix A is
[[ 1255.45872873  -718.41413243  -359.88817163 …  1025.38260481
   -513.6637602    373.04889329]
 [ -718.41413204  2158.58014172  1720.56453956 … -1194.13527399
    401.98400636  -897.54651358]
 [ -359.88817123  1720.56453939  1605.42339669 …  -798.26995276
    144.57037194  -783.63724997]
 …
 [ 1025.3826046  -1194.13527413  -798.269953   …  1208.3547055
   -440.81361876   610.37214533]
 [ -513.6637602    401.98400657   144.57037215 …  -440.81361887
    510.17392994  -151.45545363]
 [  373.04889304  -897.54651343  -783.63724993 …   610.37214515
   -151.4554535    585.27254052]]

Time required to invert the matrix A is
0.10899591445922852 seconds

The optimal solution x* is
[   1    2    3    4    5    6    7    8    9   10   11   12   13   14
   15   16   17   18   19   20   21   22   23   24   25   26   27   28
   29   30   31   32   33   34   35   36   37   38   39   40   41   42
   43   44   45   46   47   48   49   50   51   52   53   54   55   56
   57   58   59   60   61   62   63   64   65   66   67   68   69   70
   71   72   73   74   75   76   77   78   79   80   81   82   83   84
   85   86   87   88   89   90   91   92   93   94   95   96   97   98
   99  100  101  102  103  104  105  106  107  108  109  110  111  112
```

113	114	115	116	117	118	119	120	121	122	123	124	125	126
127	128	129	130	131	132	133	134	135	136	137	138	139	140
141	142	143	144	145	146	147	148	149	150	151	152	153	154
155	156	157	158	159	160	161	162	163	164	165	166	167	168
169	170	171	172	173	174	175	176	177	178	179	180	181	182
183	184	185	186	187	188	189	190	191	192	193	194	195	196
197	198	199	200	201	202	203	204	205	206	207	208	209	210
211	212	213	214	215	216	217	218	219	220	221	222	223	224
225	226	227	228	229	230	231	232	233	234	235	236	237	238
239	240	241	242	243	244	245	246	247	248	249	250	251	252
253	254	255	256	257	258	259	260	261	262	263	264	265	266
267	268	269	270	271	272	273	274	275	276	277	278	279	280
281	282	283	284	285	286	287	288	289	290	291	292	293	294
295	296	297	298	299	300	301	302	303	304	305	306	307	308
309	310	311	312	313	314	315	316	317	318	319	320	321	322
323	324	325	326	327	328	329	330	331	332	333	334	335	336
337	338	339	340	341	342	343	344	345	346	347	348	349	350
351	352	353	354	355	356	357	358	359	360	361	362	363	364
365	366	367	368	369	370	371	372	373	374	375	376	377	378
379	380	381	382	383	384	385	386	387	388	389	390	391	392
393	394	395	396	397	398	399	400	401	402	403	404	405	406
407	408	409	410	411	412	413	414	415	416	417	418	419	420
421	422	423	424	425	426	427	428	429	430	431	432	433	434
435	436	437	438	439	440	441	442	443	444	445	446	447	448
449	450	451	452	453	454	455	456	457	458	459	460	461	462
463	464	465	466	467	468	469	470	471	472	473	474	475	476
477	478	479	480	481	482	483	484	485	486	487	488	489	490
491	492	493	494	495	496	497	498	499	500	501	502	503	504
505	506	507	508	509	510	511	512	513	514	515	516	517	518
519	520	521	522	523	524	525	526	527	528	529	530	531	532
533	534	535	536	537	538	539	540	541	542	543	544	545	546
547	548	549	550	551	552	553	554	555	556	557	558	559	560
561	562	563	564	565	566	567	568	569	570	571	572	573	574
575	576	577	578	579	580	581	582	583	584	585	586	587	588
589	590	591	592	593	594	595	596	597	598	599	600	601	602
603	604	605	606	607	608	609	610	611	612	613	614	615	616
617	618	619	620	621	622	623	624	625	626	627	628	629	630
631	632	633	634	635	636	637	638	639	640	641	642	643	644
645	646	647	648	649	650	651	652	653	654	655	656	657	658
659	660	661	662	663	664	665	666	667	668	669	670	671	672
673	674	675	676	677	678	679	680	681	682	683	684	685	686
687	688	689	690	691	692	693	694	695	696	697	698	699	700
701	702	703	704	705	706	707	708	709	710	711	712	713	714
715	716	717	718	719	720	721	722	723	724	725	726	727	728
729	730	731	732	733	734	735	736	737	738	739	740	741	742
743	744	745	746	747	748	749	750	751	752	753	754	755	756
757	758	759	760	761	762	763	764	765	766	767	768	769	770
771	772	773	774	775	776	777	778	779	780	781	782	783	784

```
  785  786  787  788  789  790  791  792  793  794  795  796  797  798
  799  800  801  802  803  804  805  806  807  808  809  810  811  812
  813  814  815  816  817  818  819  820  821  822  823  824  825  826
  827  828  829  830  831  832  833  834  835  836  837  838  839  840
  841  842  843  844  845  846  847  848  849  850  851  852  853  854
  855  856  857  858  859  860  861  862  863  864  865  866  867  868
  869  870  871  872  873  874  875  876  877  878  879  880  881  882
  883  884  885  886  887  888  889  890  891  892  893  894  895  896
  897  898  899  900  901  902  903  904  905  906  907  908  909  910
  911  912  913  914  915  916  917  918  919  920  921  922  923  924
  925  926  927  928  929  930  931  932  933  934  935  936  937  938
  939  940  941  942  943  944  945  946  947  948  949  950  951  952
  953  954  955  956  957  958  959  960  961  962  963  964  965  966
  967  968  969  970  971  972  973  974  975  976  977  978  979  980
  981  982  983  984  985  986  987  988  989  990  991  992  993  994
  995  996  997  998  999 1000]

The vector b is
[1288148.38605397 1225895.69176978 1257473.64841123 1228835.82255788
 1224077.01871562 1212517.82389932 1240859.74005241 1253152.65647997
 1240228.57636776 1270103.38490047 1264142.82338179 1251688.46063073
 1254482.1363195  1225112.43408274 1263940.76502782 1255410.04538622
 1239677.20774742 1235349.11386752 1231038.64020359 1242211.03679888
 1231434.4916032  1274326.9892582  1255441.89928393 1197434.74024783
 1274248.85359364 1240688.57269685 1215064.08463919 1240491.20378502
 1268013.52287767 1222841.25313544 1231853.95927474 1272321.56748619
 1231225.74554436 1248468.52281984 1232173.80067627 1262733.3429281
 1242232.63935319 1238660.79503354 1218555.36147668 1265515.51709145
 1262823.83753068 1251383.99191741 1245206.85638745 1244614.05808569
 1257808.12575237 1276288.56409817 1250555.11610416 1291540.50838127
 1267898.33174838 1221586.73837982 1261762.54012527 1219170.18772827
 1247321.56926833 1248703.99586329 1249072.05168852 1215396.66210849
 1246632.51585232 1274542.09354422 1262926.11340021 1251961.20389832
 1251023.67083806 1242339.57914703 1255491.32801905 1218716.64152746
 1248261.53484672 1250685.32733516 1226141.03617422 1232635.91286844
 1272435.87572042 1250252.59078934 1242675.28187014 1236688.21629416
 1243897.67962365 1245703.51495165 1233543.19954982 1288941.57185538
 1198509.41844045 1260015.08577748 1254602.76998612 1229639.00630556
 1255115.99097223 1253405.58882617 1197644.35695534 1266849.14852753
 1254736.53656355 1229982.91288287 1248720.51392522 1229144.46429352
 1288155.68776751 1248370.74605938 1263127.8497193  1258886.07518881
 1281759.45916786 1199323.11761293 1234355.40813537 1273824.50105719
 1233894.33898138 1214474.94791131 1244721.46139526 1244794.07319837
 1245518.74462949 1225879.98228785 1274658.40637994 1241826.17857897
 1282226.21055016 1251817.69963363 1238609.12132459 1254127.06105312
 1240814.5583992  1251757.55090083 1262885.0717024  1249320.99152669
 1246609.59108422 1216496.92673508 1257602.62152679 1293186.23603885
 1294909.63188488 1252813.35505656 1272304.8774921  1251163.37527779
```

1216502.1299694	1266594.11851374	1284842.36537209	1281542.01729402
1299318.24333636	1240977.16137476	1271431.76179625	1266595.13012004
1248368.59211165	1255175.30590977	1246129.51102496	1278275.05902438
1242771.35691271	1249081.12162635	1235199.20504633	1259758.82793643
1224921.03358947	1252958.92046671	1240983.89255875	1216864.81553027
1250556.82683249	1281189.66226206	1184858.67823268	1244733.86092426
1240304.03386651	1227183.69727283	1268157.53192575	1251992.55711503
1221678.88880842	1221405.35449494	1229307.99907886	1290766.17083046
1279512.71764187	1218397.9016137	1254437.0508617	1246750.86567303
1232762.78964127	1264343.55922994	1215153.31614295	1266446.41736942
1228031.43709566	1233881.55471041	1272917.14523503	1286065.26981243
1289965.17220237	1266871.57380324	1269237.70700397	1221383.35448964
1260886.75002073	1216883.7180023	1227148.79668091	1214490.31047607
1261017.20865857	1274011.54610594	1272537.12011991	1266013.70244475
1255996.57321239	1220372.30196754	1268530.37470999	1242853.69042061
1247228.78172255	1271952.99049685	1270383.85926534	1248362.5400559
1235689.86112191	1234768.4327464	1252785.35494867	1277294.28686335
1223946.72683315	1271139.51315764	1301439.42399341	1295616.16490768
1277138.31795833	1267608.71138116	1237903.16495186	1236189.82438676
1258485.12675603	1251790.22731443	1215532.97640799	1226533.89719483
1260265.69526796	1271118.70727439	1259532.67557845	1232732.33843115
1304059.31458025	1236623.70012008	1242528.06856172	1172393.24853974
1275976.32647567	1240407.44007277	1249410.00402433	1243149.5177758
1250196.82447626	1286325.55653279	1267491.50547619	1234041.0290686
1279281.27110331	1233548.84881058	1222760.28334588	1272223.51269752
1227406.22189382	1269960.93009174	1236736.75815412	1253738.37032254
1262348.45044967	1269572.38955447	1227077.12501535	1257042.6819107
1292676.72048689	1268212.80874874	1248086.76910734	1257386.8081849
1242718.97451512	1258640.71810001	1296306.82616005	1248734.47094903
1249945.29298934	1232147.84699013	1262146.18416055	1258018.27856888
1230750.6592256	1258429.85204248	1241444.45031582	1253708.54880839
1285090.91176266	1238911.97973709	1209907.46591135	1250971.25334476
1238413.38206189	1273914.43962971	1223802.96269878	1221124.13984412
1267101.86327472	1229618.51092038	1252515.1822211	1234525.02044614
1208245.91706238	1256145.13028559	1250261.60286728	1223804.98907268
1278097.11101298	1240694.14223884	1210765.58377228	1268829.43452842
1262564.8416374	1274491.13675429	1224376.20619838	1267864.12725726
1228178.50700185	1236010.57531054	1204678.82022509	1297168.63898097
1200006.6590107	1262933.79757168	1239398.05232958	1264254.09524558
1218808.94068558	1291656.35429613	1248434.58276395	1242418.66414322
1254106.02257091	1254620.19861206	1268718.70593357	1209394.8671041
1263059.19826735	1270006.04317464	1257802.22733472	1257107.38544548
1224160.93340931	1242492.20730925	1245780.01598171	1282355.28405497
1277674.43517594	1230935.4468137	1248061.53180537	1264948.5662123
1262696.62374397	1271009.88962721	1224574.02632788	1236691.44194122
1314194.79032077	1221876.8936144	1264374.03364695	1237405.42485922
1207353.30016618	1246275.88687563	1238685.16873131	1252645.4106608
1241884.43628683	1239116.95219382	1241547.44866885	1242126.74542913

1235273.60136946	1252928.02355021	1221396.30739005	1229105.21772538
1256754.39277019	1274850.06530805	1261323.13840957	1188378.0161302
1245447.53620255	1282093.00769176	1240589.93557341	1258701.12179004
1252097.29425819	1274120.63519241	1268636.87219648	1202851.39112904
1267945.41645865	1250464.37733621	1273609.30841394	1262154.45244497
1246159.79394133	1221126.1528003	1271762.17048139	1313355.10549394
1239584.77330384	1260047.26083497	1243480.11491911	1242994.37654698
1264899.79377565	1264143.40431914	1233436.02420213	1289018.53880881
1262927.2144792	1238376.61012905	1268195.84554748	1260909.90019984
1249093.39579474	1259434.37032047	1270478.11961436	1284768.4910855
1260714.16486723	1266602.53059037	1271370.56699026	1295191.41536075
1238506.42362597	1236506.74990968	1252364.359876	1289696.07249361
1228962.45068579	1287934.743591	1313621.75153134	1230787.88811854
1282825.43414448	1277957.82444456	1235996.77003489	1291002.0424181
1244481.45069608	1288789.17326532	1259445.09458831	1245336.92308677
1284929.04672351	1208564.02076297	1182770.67056972	1260838.20185233
1234466.45908047	1242215.64756359	1249396.44977504	1270782.41378366
1237564.87746142	1261267.74468841	1219820.78732948	1227789.92233363
1253110.15407578	1213973.51953339	1268622.29653112	1265935.05183215
1229732.11637673	1257733.13970215	1250572.43673205	1223184.56642494
1252707.94655229	1268008.35726278	1226207.94192199	1263512.1047711
1272924.69791467	1275375.08893379	1250047.77303672	1261461.29060632
1258911.45815729	1251273.85519319	1195553.85761398	1280370.71867481
1250000.58710629	1222522.3938128	1230577.09782492	1236747.30776538
1240377.35726434	1259038.13704214	1257962.42027451	1241329.67081323
1223242.77696993	1232731.85644091	1235870.96165095	1243036.40578808
1258397.82984641	1233644.70718474	1218862.17699132	1237732.65233095
1196913.58379154	1252675.89601897	1262354.72146566	1254710.71105313
1227620.26906433	1268647.28232259	1211538.25265642	1206666.576163
1266450.91303276	1284354.29965631	1211666.79516062	1236019.82780389
1234969.35138476	1234157.87878385	1278254.56600724	1274310.3133183
1236055.87236372	1223626.90364867	1219536.14117232	1287071.56218837
1254072.42067546	1228100.21936696	1280041.93377967	1236308.34263204
1229637.00482491	1278299.48981742	1259067.25887519	1281447.16263617
1256060.19796073	1244545.97861372	1260965.45208039	1251386.78444772
1266935.60836968	1248803.93374928	1262060.63292909	1305499.9877648
1288318.3495364	1241521.82608994	1243296.49776293	1244522.81645964
1285515.10266149	1259877.83821428	1268334.23691806	1266732.20197077
1265389.9368838	1275249.77591698	1244633.84514851	1253145.00387844
1231181.49624673	1219223.72480267	1251159.94490374	1238652.12811548
1261014.10405057	1271282.91383743	1192787.65531129	1257745.24581561
1248776.33701906	1276737.03822593	1251249.01349079	1280610.23077484
1265105.31828624	1257442.50215618	1261008.26592421	1250584.96830424
1236799.67358856	1246452.8487509	1265392.88298353	1262702.89581991
1255208.99712362	1291160.598057	1239983.08375465	1290812.4890042
1247248.65796736	1247113.6139629	1224563.23158919	1224253.40546635
1270611.90446618	1263800.51326276	1238590.68780095	1213274.81664778
1245237.7471099	1253136.24299191	1220835.86841669	1261460.64451903

1269121.18957004	1300142.92846873	1262397.90113126	1299920.76887955
1225085.613558	1268557.23866189	1259351.51606394	1251629.00381079
1271629.07068972	1245591.58674152	1274875.06308414	1263839.08265705
1241629.93686851	1297119.30353206	1227264.43443037	1286680.51589303
1256041.26349824	1249112.16557615	1229170.97471523	1262215.37119532
1256858.610425	1259654.18973439	1236451.55377774	1279317.07071294
1267134.20049693	1265818.54634735	1255368.67504331	1252188.77245844
1226629.24088805	1289485.27998856	1251399.95865276	1281482.87889633
1301177.30974446	1224147.57081232	1248682.70253872	1258741.45289322
1239792.98181906	1225250.82081046	1227919.85180507	1257059.37430017
1226451.68723964	1265122.56574058	1262519.21878914	1260360.72102022
1260788.14180888	1254885.02400722	1246293.62044665	1267583.32147705
1240019.48767079	1267029.20087645	1241364.6949583	1234580.79770448
1257917.09186031	1224479.48108787	1214697.03560595	1231311.81904943
1274296.95847892	1245833.91208851	1271688.60842807	1246460.86571022
1272595.72145915	1290602.51051777	1288346.85736724	1286097.35148451
1221512.45593669	1262852.86220821	1234223.64007496	1228549.73025693
1231741.69358397	1270385.08395307	1238597.57395781	1275762.23454384
1257829.33770838	1287149.91266768	1231732.66875521	1262752.4628989
1237750.47421183	1223373.26960318	1248580.27739612	1235718.22077581
1262706.59225343	1260072.38429481	1273915.689672	1247888.83267487
1234709.9078416	1272494.19602992	1236035.21813257	1274211.48878933
1288081.30950625	1268079.5636599	1269195.03220781	1262713.99985939
1255891.01724234	1271562.92128194	1267354.74408223	1261998.64618976
1229720.43584696	1244800.04763114	1223647.85161048	1243748.74436189
1236328.36144872	1260744.26664209	1262960.25822	1228219.44678089
1250394.07896046	1226200.90889894	1264060.84755961	1284791.58306888
1292954.60300365	1219959.82768694	1270031.87389793	1218516.50037018
1245541.80159519	1224739.4531711	1226917.29267761	1253985.79932055
1246708.16879006	1238269.6390089	1282545.28253321	1263926.00487986
1218798.77654493	1238270.47110335	1251274.27985257	1236210.68671569
1243065.0822562	1254186.77457497	1263821.77276191	1228478.39443806
1296057.55138012	1267800.53207537	1283820.64479045	1211639.04892576
1263011.3390695	1305419.16038563	1214179.20641081	1184946.87225684
1302696.73186111	1236136.64440546	1230742.54315935	1236799.96106654
1214571.8952265	1273165.81833356	1229365.40078885	1228219.30992277
1254090.5168344	1257858.97047291	1268933.37944059	1263479.19302072
1245272.54493275	1235357.40512261	1267812.76083123	1269412.19716108
1271393.84628641	1256786.49158545	1247222.08396963	1277452.08505048
1293369.99136995	1266963.86845388	1260252.52744899	1246731.56238348
1244519.55133851	1222389.91458512	1289538.64049871	1248081.84206897
1274835.71416089	1235265.30999864	1288922.57716687	1235119.36609891
1228151.93218044	1270875.32926422	1225735.38246719	1284802.4512106
1264250.0629708	1224626.05720994	1244471.22901325	1306875.19232855
1275959.18853026	1282100.97119422	1220365.76695914	1267361.18674554
1301647.22095698	1264746.91383402	1241834.42630902	1279834.43370326
1216298.10467397	1212704.73369374	1271727.19562317	1225514.19912666
1276052.87892828	1254792.54270512	1234643.74937275	1262424.98279968

1293888.85223941	1258014.3667922	1289682.03201557	1239178.28707911
1233883.10830769	1254718.9496822	1246420.72118595	1263681.32026403
1225083.77925736	1283408.79158864	1298413.92172854	1233832.29648222
1273266.22990474	1265880.0699782	1259432.63152677	1285624.4478673
1220115.17085939	1254589.27225358	1284778.96664139	1249883.49313481
1254212.58258755	1228451.65769105	1244232.21864583	1268511.58305762
1259916.21331939	1289398.55127066	1225596.88914753	1258559.63763227
1291151.32481121	1265268.82855102	1250695.6057002	1233849.31042759
1267137.14939784	1242429.68447134	1275977.44034756	1245390.03212494
1200248.66361541	1285526.75707397	1239279.26929402	1238483.90004801
1198404.83018632	1234428.15771315	1273895.09487448	1276450.02733179
1270319.37777289	1244000.47644541	1259106.29348648	1276420.1272592
1256641.01047466	1248340.58413014	1247198.40579951	1260366.75925457
1270982.31302113	1258198.48159406	1266729.53873361	1257504.91047549
1229353.33114747	1258461.16787329	1231864.32377757	1263094.54855108
1231523.72904914	1243083.0163433	1232235.4280902	1269298.92607589
1257227.42185459	1248399.33613159	1259888.21342264	1252577.14249839
1205009.7221024	1266025.13968511	1271345.78667423	1266594.6592641
1226735.06427778	1265651.69849459	1228859.02042378	1246164.21717467
1218997.64389912	1285405.90906713	1252670.12276036	1255996.04607817
1272425.42474948	1238988.52131045	1245544.27839291	1235746.02708826
1240035.02379794	1234669.9370886	1252709.55619647	1312327.41276684
1235968.07474794	1279645.8850687	1258668.37138013	1237855.07854148
1235398.27925482	1254558.89072886	1227557.24130694	1249941.160861
1277838.64706301	1225603.86237252	1243905.8864942	1248065.20787803
1254061.86178811	1274551.24747246	1267536.91636217	1259093.59667718
1277575.40917413	1228774.28083896	1261612.88581976	1220024.4212725
1234051.20923597	1280300.68479355	1260126.05629174	1224668.74422654
1296527.48264839	1245445.88491708	1252394.49254924	1256876.3758758
1266155.16749556	1246105.27189301	1210879.89929602	1229747.59383089
1272455.02219419	1241499.58844458	1213966.28086063	1288384.16014354
1223163.07271593	1231276.64037043	1254363.85876456	1221705.81535907
1258093.19123552	1262932.36093885	1282871.14920801	1217659.02216614
1279311.64641878	1256219.46270226	1264366.63963124	1238722.98198521
1268715.34285418	1272818.82912743	1269830.12641074	1254260.64914818
1265074.89846676	1265347.94955218	1280944.73104552	1244750.34472092
1281726.70647335	1254383.9664416	1261206.52231204	1257970.84170977
1291577.79193522	1264514.73349441	1264583.85802961	1242706.14971418
1239807.6453735	1289004.0243354	1251579.75222244	1269095.25285501
1249856.03096322	1234280.84271489	1272699.33692985	1247687.2415828
1264759.88030047	1224671.45372066	1241619.8138245	1207199.98567216
1278529.70909054	1235218.59317395	1227141.47852653	1289236.52388844
1259484.31756196	1240494.53703008	1262456.89489684	1230287.23308346
1278655.61623773	1247059.6835172	1245356.44757085	1243603.04645575
1254020.24178353	1236485.00080191	1264289.79859344	1285720.26429114
1264660.05774751	1253365.41229836	1294610.36406135	1263780.91655647
1225832.99056572	1248845.67059467	1230500.46688922	1270616.43504297
1225066.75444767	1206806.34922758	1267516.99393662	1267357.72318245

```
1235760.67375866 1274244.49339806 1232035.35511218 1233980.56099389
1287610.2784166  1244881.68028368 1291693.8162799  1227486.19272682
1259716.65752907 1255753.65681394 1218065.14379006 1258581.19019523
1238832.23755114 1276244.91522704 1246388.26650603 1271206.59842477
1287654.1617901  1251092.60859838 1272530.05519879 1260941.99511009
1271123.87850151 1257341.40655386 1222609.34433805 1262288.6854747
1259573.08328335 1253038.30367542 1256132.47670221 1233223.50256833
1252343.8962836  1239405.37775274 1245429.90473965 1233919.15179344
1234316.93599941 1246650.36771421 1247560.40684608 1271516.13370946
1236980.72339487 1274645.69036924 1249776.63072257 1306307.03103338
1242489.38723161 1258541.43559775 1255773.1395536  1226395.53634307
1229969.92471646 1210272.07534531 1263017.69506708 1279780.85152515
1279821.96720451 1227338.45417042 1246536.22005633 1242948.65699034
1267241.21299549 1251185.14008096 1241846.48784207 1236125.84739032
1261092.45736243 1273825.31457859 1256580.03590013 1205463.56642358
1241617.38711575 1220311.37870018 1268751.49141461 1248471.19238085
1269931.05115632 1238449.63692552 1241956.14705261 1255860.39936068
1277186.31162419 1289502.07949295 1239440.52554617 1213602.90847818
1280232.93550016 1241909.01273922 1259737.41245158 1260504.22450112
1250904.50600464 1256415.3416386  1245599.75897521 1207237.20376957
1257507.18365059 1258135.44099016 1273372.29308039 1266513.31057846
1264658.02014689 1314559.92172781 1266199.4960165  1248707.06948495
1288968.50443165 1248372.93101332 1248209.20455717 1209238.62229271
1273110.91817052 1263527.86468669 1238238.76836314 1283193.21899437
1284120.09101866 1264597.92512342 1290858.12761948 1266273.39810252
1269088.531476   1252150.80515203 1273900.84721462 1217694.45776639
1237361.96434144 1243072.54039695 1270027.82128568 1256126.92228949
1262283.20536914 1267764.8649733  1220009.4662269  1216843.34192144]
```

4 Helper functions

[54]:
```python
# Returns a vector, which is the result of the gradient
def gradient(A, b, x):
    return A.dot(x) - b

# Returns a scalar, which is the norm of the result above
def gradientNorm(A, b, x):
    return np.linalg.norm(A.dot(x) - b)

# Returns a scalar, which is the norm of the difference between x and x*
def differenceNorm(x, x_opt):
    return np.linalg.norm(x - x_opt)
```

```
# Returns a scalar, which is the norm of x
def norm(x):
    return np.linalg.norm(x)
```

[55]:
```
# Plotting the difference norms log ||x - x*||2
def plotDifferenceNorms(scale, maxIter, tolerance, algorithmName, algorithm,␣
↪logBase):
    A = np.load(f"data/{scale}Matrix.npy", allow_pickle=True)
    b = np.load(f"data/{scale}Vector.npy", allow_pickle=True)
    x_opt = np.load(f"data/{scale}Solution.npy", allow_pickle=True)

    print(f"\nThe {scale} scale problem is chosen. The matrix A and vector b␣
↪dimension is {b.size}")
    print(f"The number of maximum iterations is {maxIter}. The allowed␣
↪tolerance for gradient norm is {tolerance}" )

    start = time.time()
    x_opt_algo, x_iterations_algo, stoppingReason = algorithm(A, b, maxIter,␣
↪tolerance)
    end = time.time()

    print(f"\nThe {algorithmName} algorithm runs in {end - start} seconds")
    print("Reason of stopping")
    print(stoppingReason)

    if scale == "huge":
        print(f"\nFirst 100 values in the optimal solution x found by␣
↪{algorithmName} algorithm")
        print(x_opt_algo[0:100])
        print("\nFirst 100 values in the theoretical optimal solution x*")
        print(x_opt[0:100])
    else:
        print(f"\nThe optimal solution x found by {algorithmName} algorithm")
        print(x_opt_algo)
        print("\nThe theoretical optimal solution x*")
        print(x_opt)

    differenceNorms = []
    for x_sol in x_iterations_algo:
        differenceNorms.append(differenceNorm(x_sol, x_opt))
    differenceNorms = np.array(differenceNorms)

    figure(figsize=(8, 6), dpi=80)

    size = 16
    iterations = np.arange(0, differenceNorms.size, 1)
```

```
    plt.plot(iterations, differenceNorms, label = f"Experimental convergence␣
↪rate")
    plt.title(f"Convergence rate of\n{algorithmName} algorithm\n{scale} scale␣
↪problem - dimension: {b.size}", size=size + 4)
    plt.xticks(fontsize=size)
    plt.yticks(fontsize=size)
    # Plotting the log graph in base 2
    plt.yscale('log', base=2)
    plt.xlabel("Iterations", size=size)
    plt.ylabel(r'$log||x-x*||_2$', size=size)
    plt.legend(loc=1, frameon=False, fontsize=size + 2)
    plt.show()
```

5 Task 1: Gradient Descent Algorithm

5.1 Gradient descent algorithm implementation

[56]:
```
def gradientDescent(A, b, maxIters = 5000, epsilon = 10e-5):
    # Dimension of A and b
    dim = b.size
    # initial random vector x filled with the mean of matrix A, with length␣
↪equal to the dimension
    x = np.repeat(np.mean(A), dim)
    # The Lipschitz constant, which is the maximum eigenvalue of A
    L = np.max(np.linalg.eigvals(A))
    # The step size is alpha = 1/L
    alpha = 1/L
    # currentIteration
    iter = 1
    # Saving the results
    x_iterations = [x]
    # The main iteration loop
    while (gradientNorm(A, b, x) > epsilon and iter <= maxIters):
        x = x - alpha * gradient(A, b, x)
        x_iterations.append(x)
        iter += 1
    # Stopping reason (max iteration exceeded or gradient norm smaller than the␣
↪tolerance epsilon)
    if iter > maxIters:
        stoppingReason = f"Max iterations ({maxIters}) exceeded"
    else:
        stoppingReason = f"Gradient norm smaller than {epsilon}\nCompleted␣
↪iteration: {iter}"
    return (x, x_iterations, stoppingReason)
```

5.2 Gradient descent algorithm convergence rate analysis

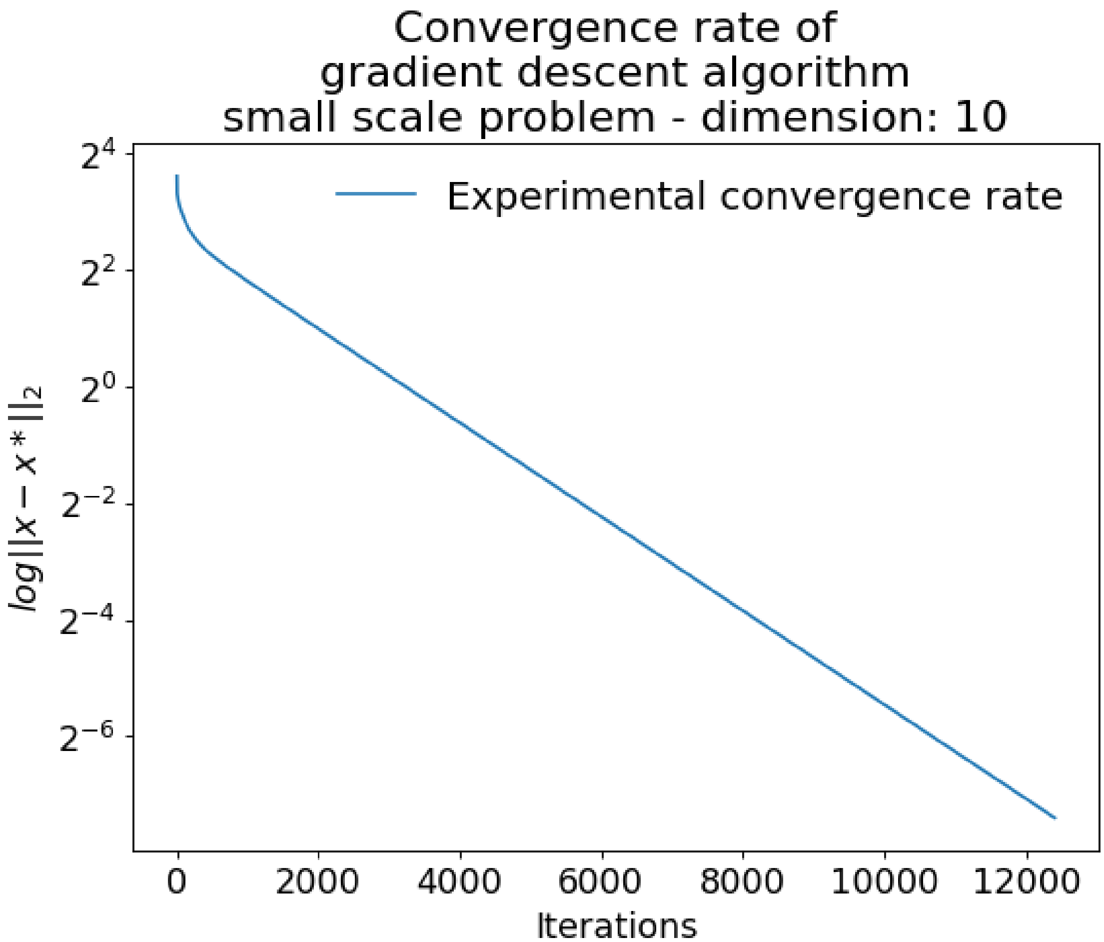
[57]:
```
# There are three different scales: small, large and huge
scales = ["small", "large", "huge"]
# Number of maximum iterations of the three scales small, large and huge
maxIters = [20000, 40000, 60000]
# Tolerance of the gradient norm of the three scales small, large and huge
tolerances = [10e-5, 10e-3, 10e-1]
# The algorithm
algorithmName = "gradient descent"
algorithm = gradientDescent
# The logarithm base
logBase = 2
# Running optimization for the three scales small, large and huge
for i in range(0,3):
    plotDifferenceNorms(scales[i], maxIters[i], tolerances[i], algorithmName,␣
 ↪algorithm, logBase)
```

```
The small scale problem is chosen. The matrix A and vector b dimension is 10
The number of maximum iterations is 20000. The allowed tolerance for gradient
norm is 0.0001

The gradient descent algorithm runs in 0.11102747917175293 seconds
Reason of stopping
Gradient norm smaller than 0.0001
Completed iteration: 12410

The optimal solution x found by gradient descent algorithm
[1.00126508 2.00079503 3.00132805 4.00138349 5.00101498 5.99769032
 7.00135383 7.99564111 8.99891374 9.99914945]

The theoretical optimal solution x*
[ 1  2  3  4  5  6  7  8  9 10]
```



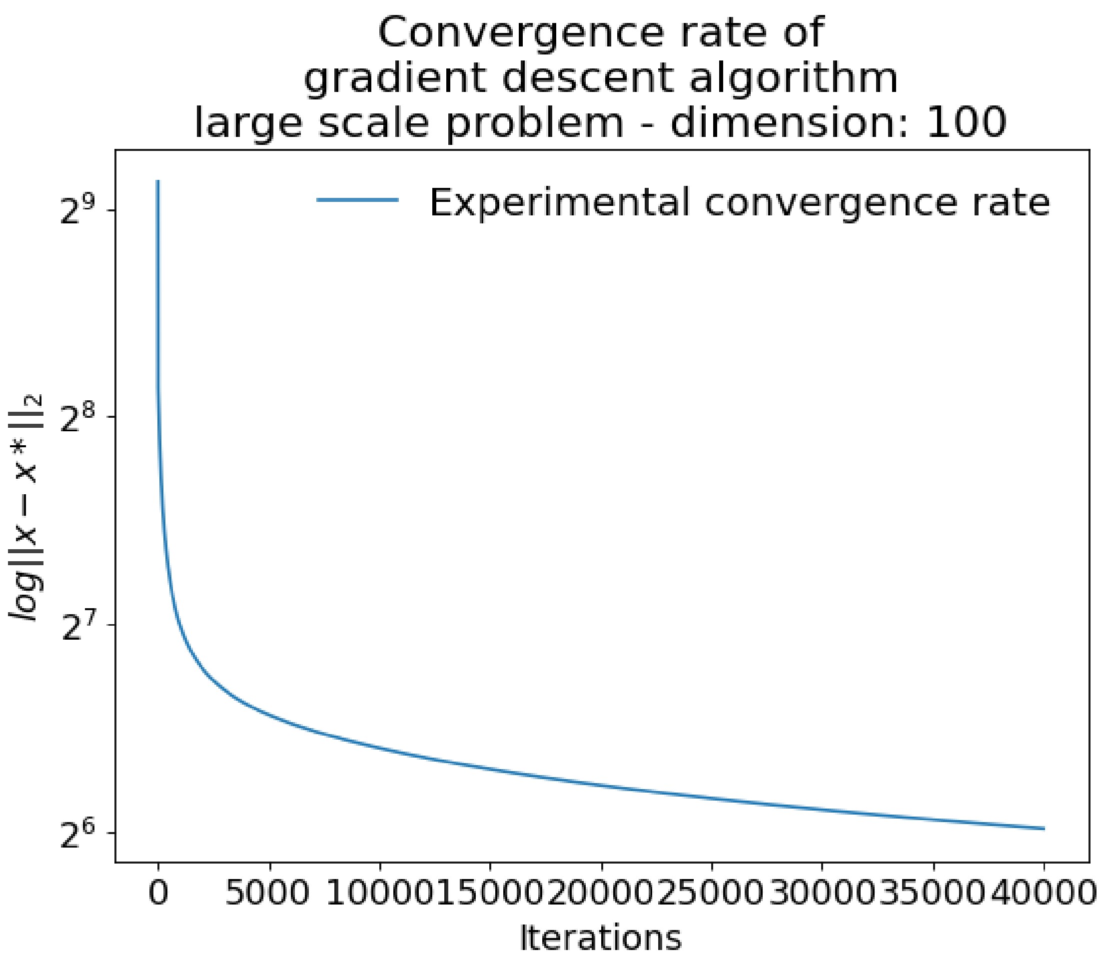

```
The large scale problem is chosen. The matrix A and vector b dimension is 100
The number of maximum iterations is 40000. The allowed tolerance for gradient
norm is 0.01

The gradient descent algorithm runs in 9.819963932037354 seconds
Reason of stopping
Max iterations (40000) exceeded

The optimal solution x found by gradient descent algorithm
[ 10.80954396  13.53080539  12.34048827  13.56010399  13.21795864
   3.44248581   2.58263856   9.56551187   2.56131046  10.9542098
   2.02486301  17.76882364  17.15608743  14.20430406  16.40691544
   8.04793459  22.89717879  33.23654019  17.13214426  23.06360859
   8.69197351  21.62729296  28.15997518  25.24696302  19.44788055
  28.4998845   31.10315707  32.34107394  23.97768023  28.70397213
  33.47529818  31.31788127  30.57097049  42.9701813   25.36785319
  44.08458021  42.57848789  49.70319644  40.79161149  43.17582171
```

```
 39.66958338  29.48079611  53.8984904   43.90625763  46.05410526
 45.78675598  37.05648298  58.2827156   54.04590147  44.93695454
 48.81566026  44.28675882  49.047388    60.45390524  61.11938949
 40.53789776  60.62875042  56.28938643  62.1533439   61.76770693
 51.48532466  67.58310946  61.32562353  62.64173655  66.11893653
 62.32141118  63.33040059  67.5100942   76.88171982  70.84572134
 68.48750716  79.83034661  67.32404466  68.93852863  76.16173337
 76.48988536  82.8672848   77.91749532  80.45729634  71.02831119
 82.99051374  85.51164549  89.89223927  83.19374159  73.46159411
 89.37990293  76.8168497   91.13527585  91.98504286  94.51285716
 76.8489968   81.06639423  85.75055827  82.75699504  92.51526601
 95.51839141  89.29033727  95.65619384 102.39389886  89.53180617]

The theoretical optimal solution x*
[  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
  91  92  93  94  95  96  97  98  99 100]
```

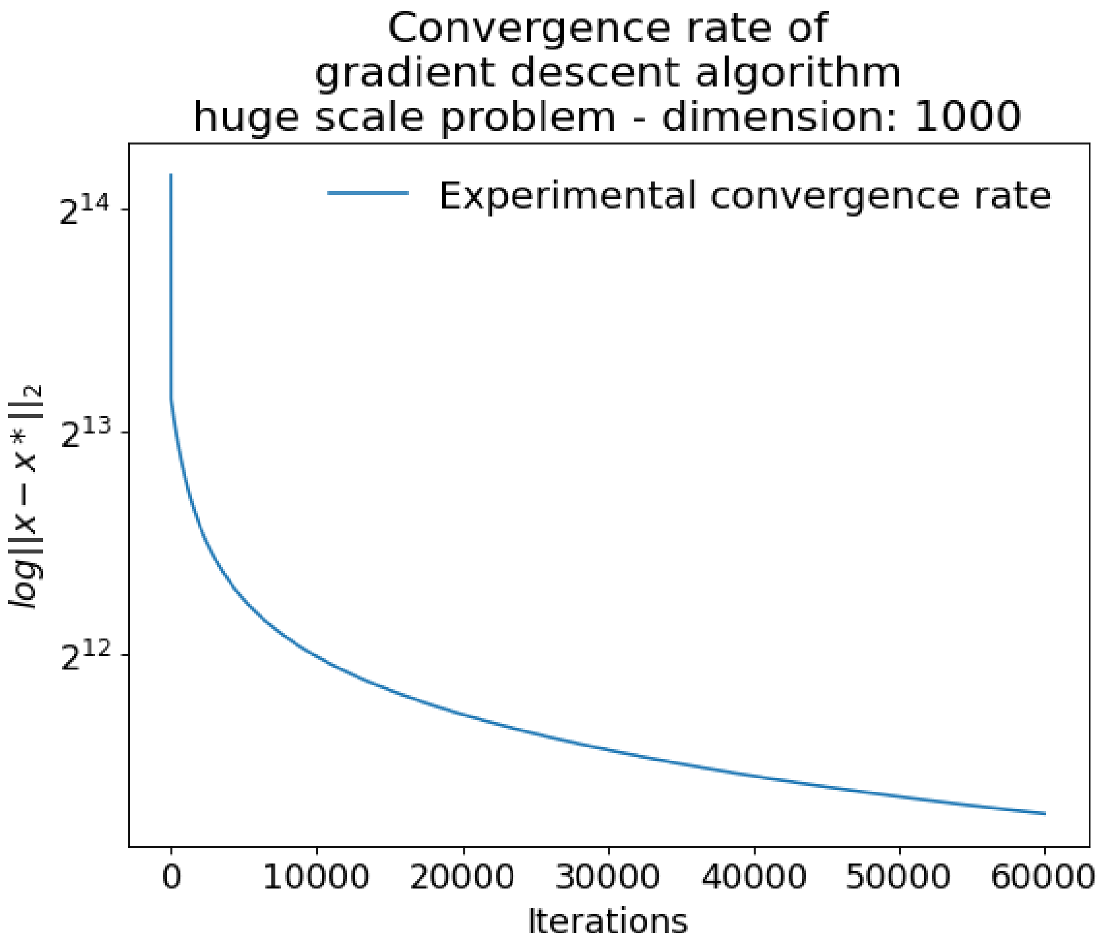


```
The huge scale problem is chosen. The matrix A and vector b dimension is 1000
The number of maximum iterations is 60000. The allowed tolerance for gradient
norm is 1.0

The gradient descent algorithm runs in 34.8260064125061 seconds
Reason of stopping
Max iterations (60000) exceeded

First 100 values in the optimal solution x found by gradient descent algorithm
[-10.63174488  63.37143999 157.67587662 -28.98040714  18.44819265
   8.89392715  91.04566023  42.1484773  106.46289367  56.4020265
  70.9809846   67.83963515  82.22141957  70.79467462 163.73643644
  17.01598209 -52.40830661  49.13319062  45.59715111 149.41525211
  43.00752814  44.7106021  -21.74792399 127.86263676  74.14244684
  79.51530711  22.52373491 -24.35930537 -18.73706392 -23.63218186
  -3.24740986 115.55521407 139.73619666 133.63803682  46.38450882
 -60.23245944 192.40687214  57.71963766  62.39052301 -38.6454732
  41.17525845  31.5090981   71.09865821  45.91672493 103.90332947
 153.50563552 -13.61070701 114.34910011 240.60198667 165.59520295
  97.3492586  147.7839864  108.61395491  83.05104859 142.45615019
  92.53214458  68.65573783  80.94962884 154.63271263 121.71339939
 167.58529837 104.6968628   11.07101534  97.61481586  89.13633389
 128.60756142 188.79503521  98.14144784  87.77590542 101.83752068
  -1.78828288 108.89490175 160.52246442 191.82613482 141.4979839
 181.23537129 190.12705397  10.9273383  224.73054108  55.03595881
  88.36713869  79.19878383 175.2532021  108.02181236 205.83864386
 159.01782838  46.4736954  134.76585137  15.2735539  149.76101792
 127.14084231  50.26364769 208.6255157  220.10366406  59.03353063
 126.83309631  40.4653204  152.97359151 266.45642179 136.7430323 ]

First 100 values in the theoretical optimal solution x*
[  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
  91  92  93  94  95  96  97  98  99 100]
```



From the three graphs, it appears that the implemented gradient descent has a sublinear to nearly linear convergence rate.

I use base 2 logarithm because the convergence rate of gradient descent only decreases linearly, as base 10 is big and the graph will not show any information on the y-axis

For the small scale problem, gradient descent returns optimal solution close to x*

For the large scale problem, gradient descent returns suboptimal solution that has the same pattern as x*

For the huge scale problem, gradient descent returns suboptimal solution that is still far from x* because the dimension is too large (1000)

6 Task 2: Conjugate Gradient Algorithm

6.1 Conjugate gradient algorithm implementation

[58]:
```python
def conjugateGradient(A, b, maxIters = 5000, epsilon = 10e-5, period = 100):
    # Dimension of A and b
    dim = b.size
    # initial random vector x filled with the mean of matrix A, with length␣
↪equal to the dimension
    x = np.repeat(np.mean(A), dim)
    # The Lipschitz constant
    L = np.max(np.linalg.eigvals(A))
    # The step size is alpha = 1/L
    alpha = 1/L
    # currentIteration
    iter = 1
    # Saving the results
    x_iterations = [x]

    gradients = []
    gradientNorms = []

    rhos = []

    while (gradientNorm(A, b, x) > epsilon and iter <= maxIters):
        if (iter % period == 1):
            gradientNorms.append(gradientNorm(A,b,x))
            gradientVector = gradient(A, b, x)
            gradients.append(gradientVector)
            rho = - gradientVector
            rhos.append(rho)
            x = x + alpha * rho
            x_iterations.append(x)
        else:
            gradientNorms.append(gradientNorm(A,b,x))
            gradients.append(gradient(A, b, x))
            delta = (gradientNorms[iter - 1] ** 2)/(gradientNorms[iter - 2] **␣
↪2)
            rho = - gradients[iter - 1] + delta * rhos[iter - 2]
            rhos.append(rho)
            x = x + alpha * rho
            x_iterations.append(x)
        iter += 1

    if iter > maxIters:
        stoppingReason = f"Max iterations ({maxIters}) exceeded"
    else:
```

```
        stoppingReason = f"Gradient norm smaller than {epsilon}\nCompleted␣
↪iteration: {iter}"
    return (x, x_iterations, stoppingReason)
```

6.2 Conjugate gradient algorithm convergence rate analysis

[59]:

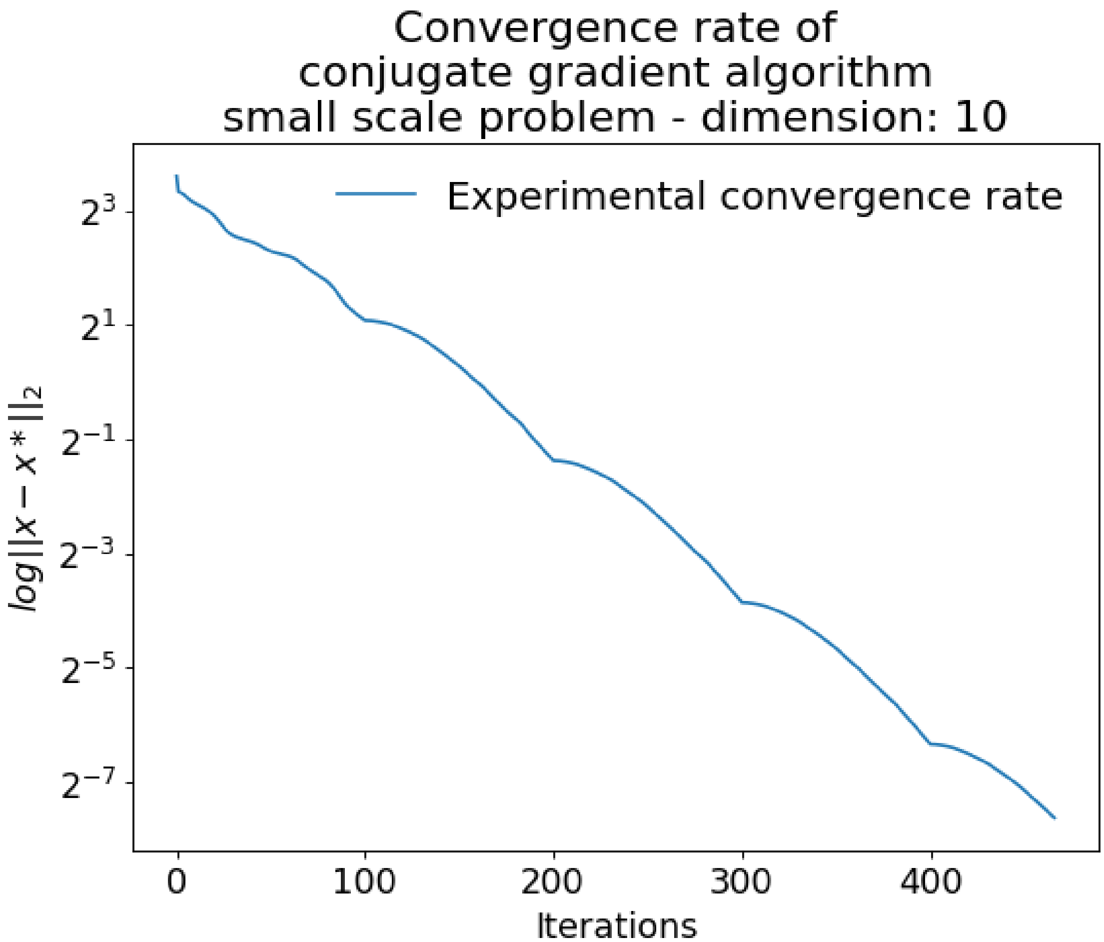
```
# There are three different scales: small, large and huge
scales = ["small", "large", "huge"]
# Number of maximum iterations of the three scales small, large and huge
maxIters = [20000, 40000, 60000]
# Tolerance of the gradient norm of the three scales small, large and huge
tolerances = [10e-5, 10e-3, 10e-1]
# The algorithm
algorithmName = "conjugate gradient"
algorithm = conjugateGradient
# The logarithm base
logBase = 2
# Running optimization for the three scales small, large and huge
for i in range(0,3):
    plotDifferenceNorms(scales[i], maxIters[i], tolerances[i], algorithmName,␣
↪algorithm, logBase)
```

```
The small scale problem is chosen. The matrix A and vector b dimension is 10
The number of maximum iterations is 20000. The allowed tolerance for gradient
norm is 0.0001

The conjugate gradient algorithm runs in 0.009025812149047852 seconds
Reason of stopping
Gradient norm smaller than 0.0001
Completed iteration: 467

The optimal solution x found by conjugate gradient algorithm
[1.00113074 2.00040352 3.0014233  4.00135845 5.00073744 5.99799636
 7.00095232 7.99627049 8.99914574 9.99935844]

The theoretical optimal solution x*
[ 1  2  3  4  5  6  7  8  9 10]
```



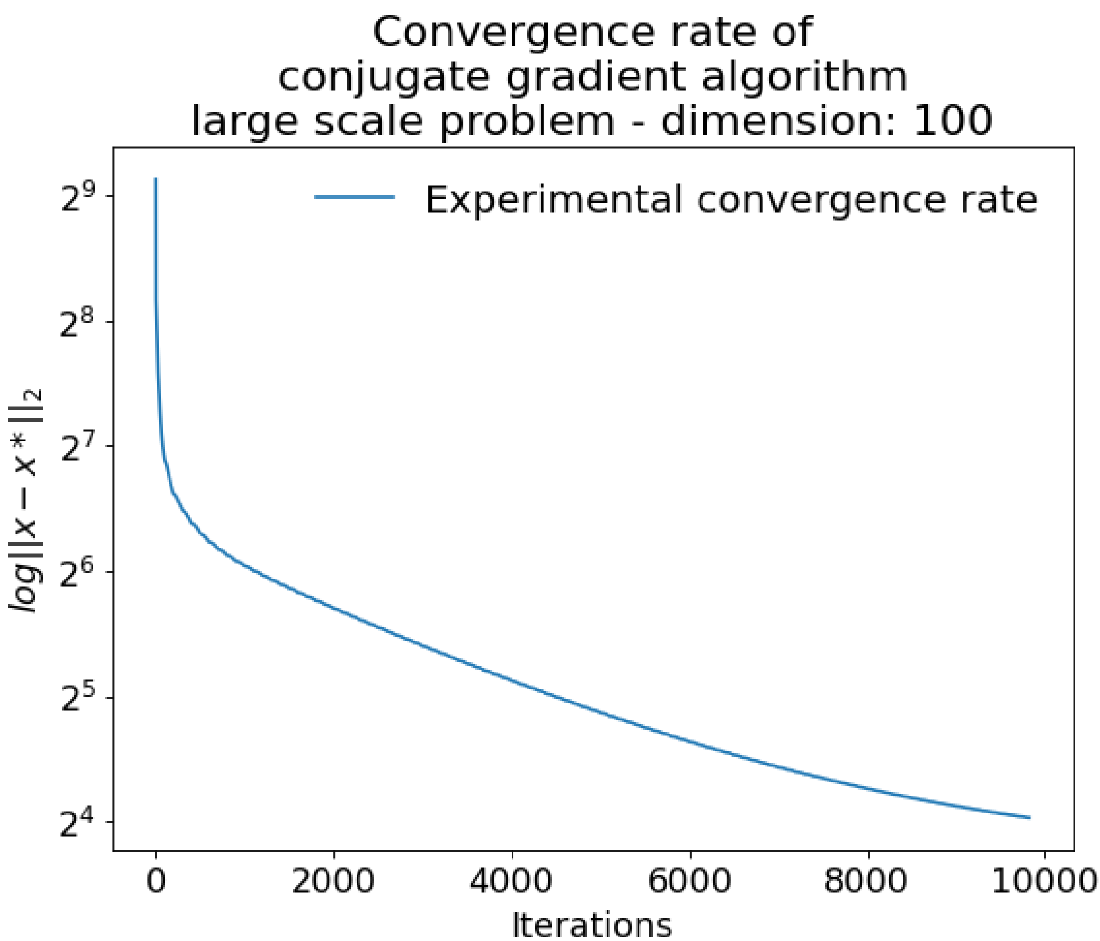

```
The large scale problem is chosen. The matrix A and vector b dimension is 100
The number of maximum iterations is 40000. The allowed tolerance for gradient
norm is 0.01

The conjugate gradient algorithm runs in 3.840991973876953 seconds
Reason of stopping
Gradient norm smaller than 0.01
Completed iteration: 9823

The optimal solution x found by conjugate gradient algorithm
[ 1.11845736  3.82449596  5.59523166  5.98512795  8.37384602  5.92841262
  8.13534334  8.18346144  9.21724186 11.52938575  7.16282571 12.55292507
  9.99412557 13.55998946 15.8113756  14.63876449 18.55548775 21.29674155
 18.01564307 20.73430025 19.27528084 20.65206849 21.40484399 23.25767325
 24.43084158 25.58133179 28.70759418 29.75083553 27.78756451 31.44894691
 32.68742246 31.56959598 34.3672904  34.67444323 32.67273952 35.97099353
 37.24588129 39.96257566 38.08324313 40.15028579 43.89269533 39.40561199
```

```
 45.94381356 43.37593535 44.08142957 46.09508346 45.84955739 49.46344864
 51.58919538 48.07725847 50.62617696 49.38565347 52.32973617 55.90379266
 55.66569541 53.74651079 55.55973899 57.54897615 60.39739274 61.23529851
 57.30760211 60.38931943 62.88273663 64.75815727 67.12739098 65.71134963
 67.20393953 68.95358952 69.47680588 72.13896062 71.84075926 72.55925089
 72.22615688 72.27349425 76.78555679 75.72601205 78.06025144 79.28783052
 79.23312947 78.84645431 79.93175    83.09755769 83.87617387 81.99399407
 82.79711543 85.13748553 85.7513178  88.2150248  91.28535614 92.50966152
 87.40856966 89.53616214 89.92482951 92.74008419 94.46197493 96.9215144
 95.15131279 97.24234619 99.95992433 99.46343368]

The theoretical optimal solution x*
[  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
  91  92  93  94  95  96  97  98  99 100]
```

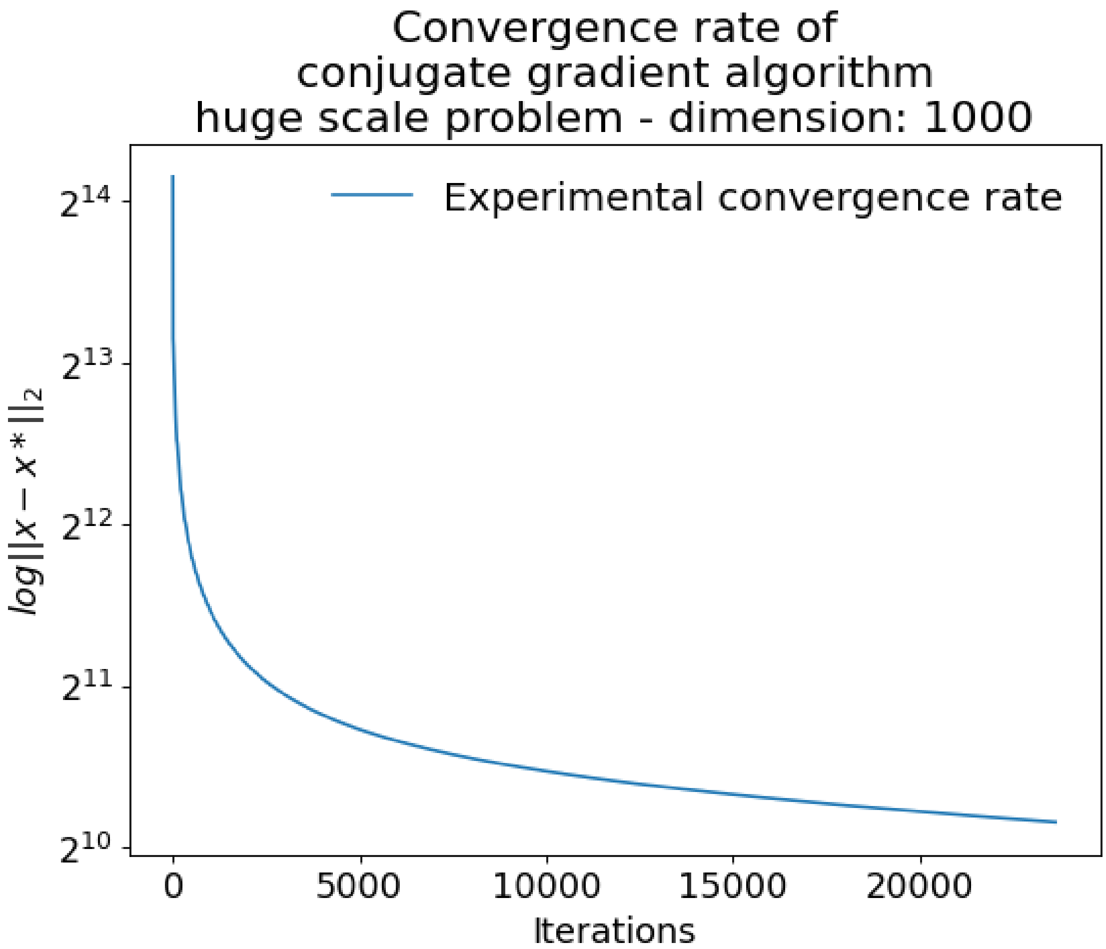


```
The huge scale problem is chosen. The matrix A and vector b dimension is 1000
The number of maximum iterations is 60000. The allowed tolerance for gradient
norm is 1.0

The conjugate gradient algorithm runs in 22.06999945640564 seconds
Reason of stopping
Gradient norm smaller than 1.0
Completed iteration: 23630

First 100 values in the optimal solution x found by conjugate gradient algorithm
[-3.32597261e+01  2.67394699e+01  3.65614952e+01 -1.66945474e+01
  5.76708378e+00  3.58056494e+01 -1.59294870e+01  2.36735775e+01
  1.97478852e+01  5.82344457e+01  2.99316448e+01 -1.91287839e+01
  2.75501216e+01  1.61975741e+01  5.30509243e+01  7.93120223e-02
  1.73846033e+01  5.95173660e+00  5.26046964e+01  8.68809282e+01
  2.31772120e+01 -1.91380419e+01  1.73136072e+01  3.51953234e+01
  1.25673999e+02  3.15630144e+01 -7.46888261e+00  1.66473683e+01
  4.20094226e+01  3.72441865e+00 -1.91348219e+01  5.74741411e+01
  9.67722766e+01  6.69688270e+01  6.33407970e+01  2.21848958e+01
  9.39510118e+01  1.86679066e+01  5.59229062e+01  4.88220892e+01
  7.29302020e+01  2.64264203e+01  1.39717552e+01  4.15625854e+01
 -2.12819961e+00  6.33192822e+01 -3.43799578e+01  6.68174526e+01
  9.18657530e+01  8.82029264e+01  5.46540702e+01  9.95163880e+01
  4.46939205e+01  2.14902166e+01  7.38928585e+01  2.17458532e+01
  7.12783135e+01  5.27827875e+01  3.99669150e+01  6.01247285e+01
  9.34323829e+01  1.04346879e+02  2.34800243e+01  2.48187161e+01
  4.22327986e+01  1.04362547e+02  7.90315636e+01  4.89505753e+01
  6.47104836e+01  9.93002052e+01  2.72200086e+01  1.95567312e+01
  7.93460548e+01  8.49954654e+01  6.51297542e+01  1.19231358e+02
  4.49252558e+01  9.74449310e+01  1.26617145e+02  8.64696118e+01
  7.20525822e+01  7.69059731e+01  1.12038938e+02  8.23099986e+01
  1.47370210e+02  1.33486088e+02  8.88785454e+01  1.19244692e+02
  5.78003204e+01  1.41824208e+02  9.58324929e+01  5.55414266e+01
  9.22953647e+01  1.12927704e+02  9.83759588e+01  7.84487926e+01
  7.10830502e+01  1.74305455e+02  1.16923181e+02  9.17655116e+01]

First 100 values in the theoretical optimal solution x*
[  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
  91  92  93  94  95  96  97  98  99 100]
```



From the three graphs, it appears that the implemented conjugate gradient has a superlinear convergence rate.

For the small scale problem, conjugate gradient returns optimal solution close to x*

For the large scale problem, conjugate gradient returns a solution whose value has an increasing trend like x*

For the huge scale problem, conjugate gradient returns suboptimal solution, although it still has an increasing trend like x*

By increasing trend, I mean I have deliberately chosen x* to be increasing natural numbers to easily test the result returned by the algorithms

The conjugate gradient also shows noticeable turbulence during the early iterations where the errors start to reduce significantly

7 Task 3: FISTA algorithm

FISTA is acronym of "Fast Iterative Shrinkage-Thresholding Algorithm"

7.1 FISTA algorithm implementation

[44]:
```python
def FISTA(A, b, maxIters = 5000, epsilon = 10e-5):
    # Dimension of A and b
    dim = b.size
    # initial random vector x0 filled with the mean of matrix A, with length␣
↪equal to the dimension
    x = np.repeat(np.mean(A), dim)
    # assign y1 equals to x0
    y = x
    # assign t equals to 1
    t = 1
    # currentIteration
    iter = 1
    # Saving the results
    x_iterations = [x]
    y_iterations = []
    t_iterations = []

    gradients = []

    # The Lipschitz constant
    L = np.max(np.linalg.eigvals(A))
    # The step size is alpha = 1/L
    alpha = 1/L

    while (gradientNorm(A, b, x) > epsilon and iter <= maxIters):
        # For example, this is the first iteration, where k = 1 (iter = 1)
        # Saving the previous x, which is x0
        previous_x = x
        # x is now x1, y ix now y1
        x = y - alpha * gradient(A, b, y)
        x_iterations.append(x)
        # Saving the previous t, which is t1
        previous_t = t
        # t is now t2 and the latter t is still t1
        t = 1/2 * (1 + math.sqrt(1 + 4 * (previous_t ** 2)))
        # y is now y2, x is x1 and x_previous is x0
        y = x + (previous_t - 1)/t * (x - previous_x)
        iter += 1

    if iter > maxIters:
        stoppingReason = f"Max iterations ({maxIters}) exceeded"
    else:
        stoppingReason = f"Gradient norm smaller than {epsilon}\nCompleted␣
↪iteration: {iter - 1}"
```

```
    return (x, x_iterations, stoppingReason)
```

7.2 FISTA algorithm convergence rate analysis

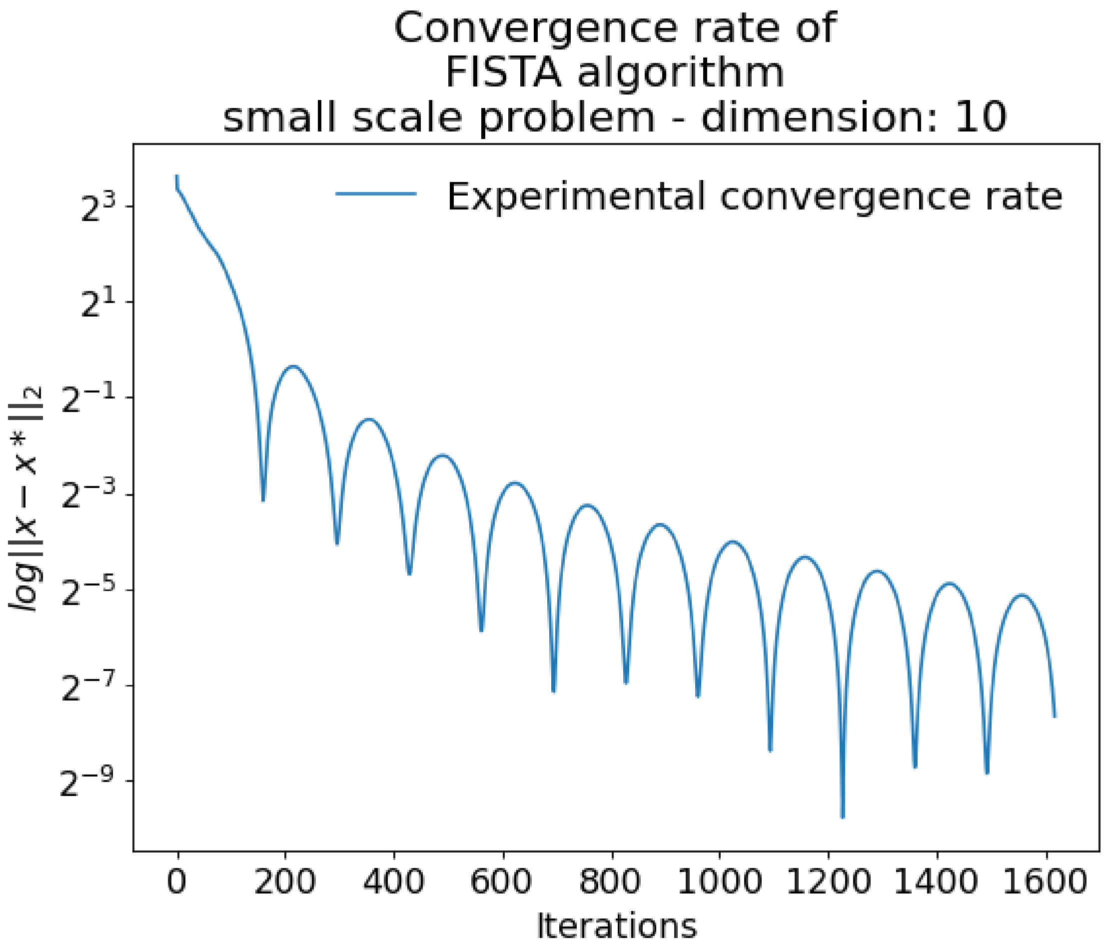
[60]:
```
# There are three different scales: small, large and huge
scales = ["small", "large", "huge"]
# Number of maximum iterations of the three scales small, large and huge
maxIters = [20000, 40000, 60000]
# Tolerance of the gradient norm of the three scales small, large and huge
tolerances = [10e-5, 10e-3, 10e-1]
# The algorithm
algorithmName = "FISTA"
algorithm = FISTA
# The logarithm base
logBase = 2
# Running optimization for the three scales small, large and huge
for i in range(0,3):
    plotDifferenceNorms(scales[i], maxIters[i], tolerances[i], algorithmName,␣
 ↪algorithm, logBase)
```

```
The small scale problem is chosen. The matrix A and vector b dimension is 10
The number of maximum iterations is 20000. The allowed tolerance for gradient
norm is 0.0001

The FISTA algorithm runs in 0.021035194396972656 seconds
Reason of stopping
Gradient norm smaller than 0.0001
Completed iteration: 1618

The optimal solution x found by FISTA algorithm
[ 0.99882646  1.99899609  2.99909342  3.99902631  4.99919153  6.00173722
  6.99881059  8.00366691  9.00111356 10.00085021]

The theoretical optimal solution x*
[ 1  2  3  4  5  6  7  8  9 10]
```



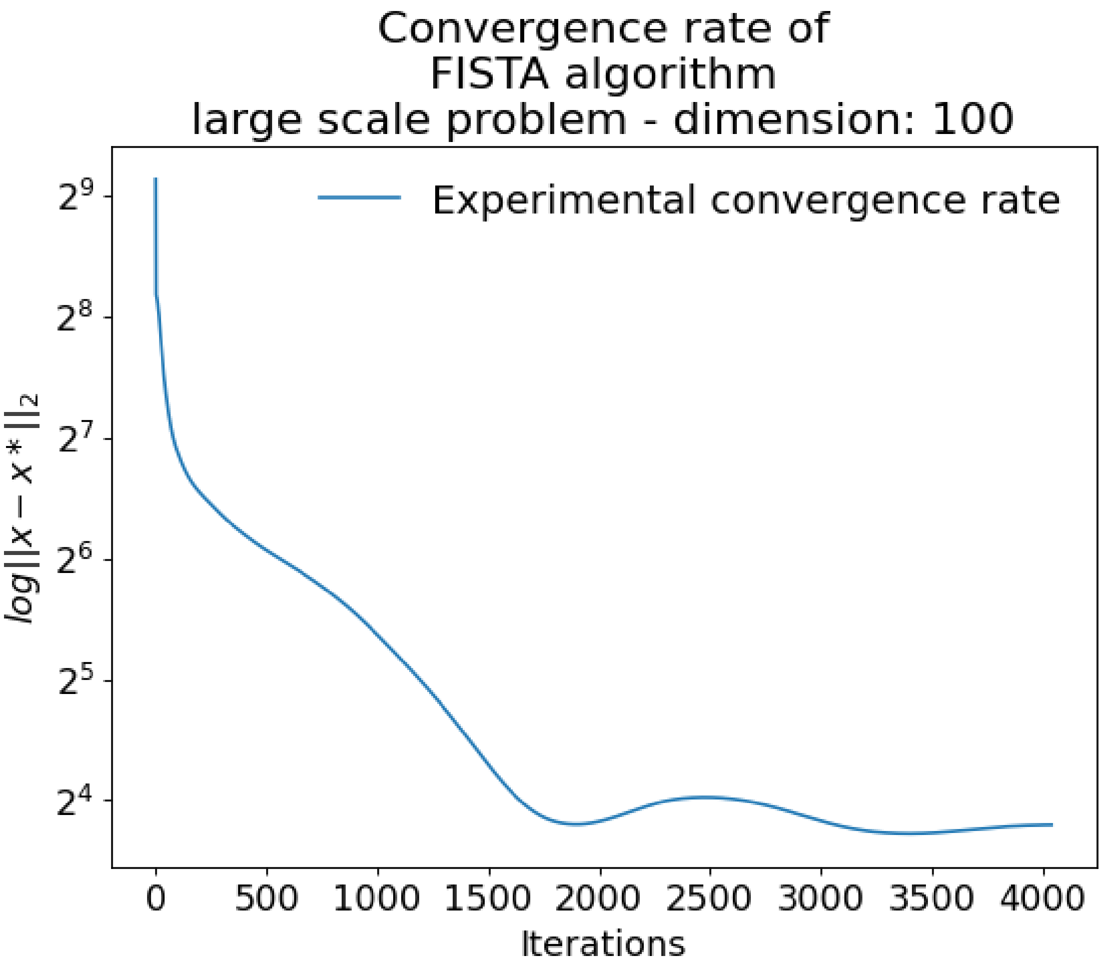

```
The large scale problem is chosen. The matrix A and vector b dimension is 100
The number of maximum iterations is 40000. The allowed tolerance for gradient
norm is 0.01

The FISTA algorithm runs in 1.0489630699157715 seconds
Reason of stopping
Gradient norm smaller than 0.01
Completed iteration: 4038

The optimal solution x found by FISTA algorithm
[  0.5804197    3.49937851   4.89739875   5.43814026   8.2733384
   6.06932763   8.32561155   8.58557714   9.73348927  10.82446087
   7.71904481  11.20116919   9.67236023  13.36241332  15.35118423
  14.87964507  18.04439859  20.17234125  18.63427131  21.34265901
  20.38616586  21.05935953  21.35136413  22.7698276   24.30630488
  25.21044793  29.10143877  29.66149567  27.76431124  31.91013713
  32.62781383  31.5158694   34.72057296  33.51675469  33.61008684
```

```
  35.43708197  36.33179422  39.16351096  37.97249797  40.45407114
  43.7780257   40.50797794  44.62097971  43.4433036   43.96178159
  46.02935254  46.97377021  48.10345962  50.94346539  48.11515001
  50.84876113  49.6817762   53.09498024  55.95326384  54.91944472
  54.82495121  55.21809438  58.09844605  60.46227026  60.56368964
  58.39584443  59.86170489  63.25100034  64.80830732  66.97287797
  65.72819593  67.79311612  68.50607027  69.32864347  72.1111604
  72.09514606  71.7748174   72.73075049  72.75332264  76.26977995
  75.97086927  77.44400962  79.29045818  79.3294741   79.52988738
  79.95864334  82.40012622  83.03507198  81.83079594  82.85343074
  84.56631005  85.96829517  88.76676183  91.23441379  91.96725468
  88.30332248  90.51121035  90.4400783   93.8698661   94.96714562
  96.84842306  95.92304513  97.5761641  100.18765082  99.86925561]

The theoretical optimal solution x*
[  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
  91  92  93  94  95  96  97  98  99 100]
```



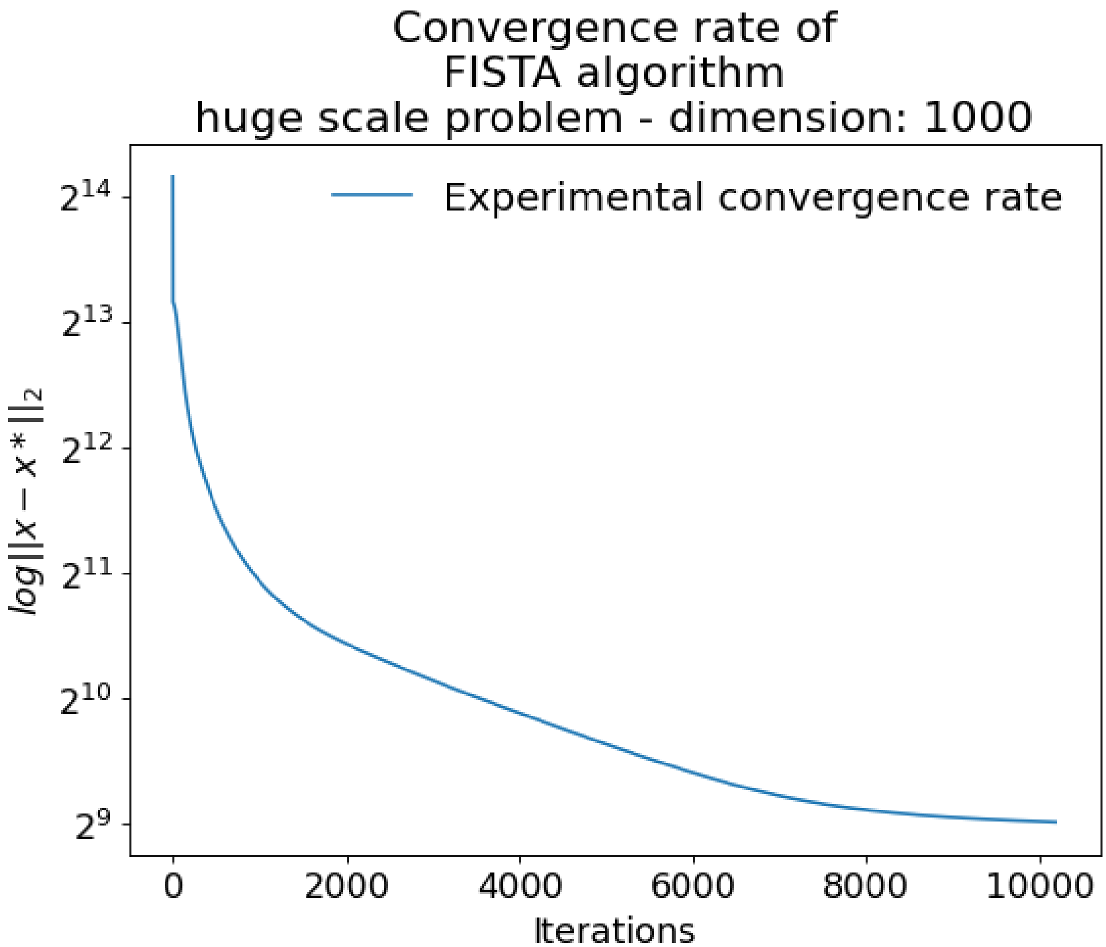

```
The huge scale problem is chosen. The matrix A and vector b dimension is 1000
The number of maximum iterations is 60000. The allowed tolerance for gradient
norm is 1.0

The FISTA algorithm runs in 8.035976886749268 seconds
Reason of stopping
Gradient norm smaller than 1.0
Completed iteration: 10184

First 100 values in the optimal solution x found by FISTA algorithm
[  3.17758067  13.90709143   6.2257751   12.74674775  -9.2641401
  25.04514651   1.18279701  11.11102729  -1.00895173  50.32358794
  32.52995236   7.83888725  15.39044038   9.99965363  15.17581813
  -6.08194271  39.45570317  22.68464105  27.96035342  33.06690366
  20.47455608   9.90012816  18.38430427  39.78134304  28.19009002
  50.51040477  26.38980085  54.63117852  45.14140792  31.76175477
```

```
  20.50274681  23.66084756  67.93184858  43.61313347  37.79765888
  -4.35895841  51.81728362  41.89768425  35.45183519  46.97417003
  29.65361469  36.69807236  68.0127728   36.24465202  23.11418608
  33.22279616  40.23792797  43.87409123  58.45726856  68.22955575
  58.75432045  87.97692044  42.83803203  75.64180086  61.89088511
  69.27824164  36.82995876  45.11893975  38.04609984  39.35089089
  58.3395861   57.85263588  56.70727358  37.46462927  75.12688082
  70.72628701  63.43135767  78.83189416  69.61501011  88.20094158
  44.76383585  65.59348403  41.42195482  67.78668395  54.35791682
  82.6495478   58.77283997  95.346273   100.99343729  77.5348936
  80.272615   114.00730891  70.52956246  87.26150746  92.56950051
 104.87676885  99.67191676  77.24307112  82.37433418  99.40422848
  85.03055606  87.16685688 114.83056539  81.10002986 105.36691563
 103.84455875  94.18861197  88.77848881 107.25604763  97.46812361]

First 100 values in the theoretical optimal solution x*
[  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
  91  92  93  94  95  96  97  98  99 100]
```



From the three graphs, it appears that the implemented FISTA has a superlinear convergence rate.

For the small scale problem, FISTA returns optimal solution close to x*

For the large scale problem, FISTA returns optimal solution that is also close to x*

For the huge scale problem, FISTA returns suboptimal solution again, because the dimension is too large (1000)

FISTA has unique pattern, whose errors rise and fall in a periodic manner. However, this behavior is only observed in small dimension problems (10). For large dimensions like 100 and 1000, FISTA doesn't seem to show this periodic error behavior

8 Task 4: Coordinate Descent Algorithm

8.1 Deterministic (cyclic) coordinate descent algorithm implementation

[46]:
```
def coordinateDescent(A, b, maxIters = 5000, epsilon = 10e-5, period = 100):
    # Dimension of A and b
    dim = b.size
    # initial random vector x filled with the mean of matrix A, with length␣
 ↪equal to the dimension
    x = np.repeat(np.mean(A), dim)
    # The Lipschitz constant
    L = np.max(np.linalg.eigvals(A))
    # The step size is alpha = 1/L
    alpha = 1/L
    # currentIteration
    iter = 1
    # Saving the results
    x_iterations = []

    while (gradientNorm(A, b, x) > epsilon and iter <= maxIters):
        for index in range(0, dim):
            x_totalGradient = x - alpha * gradient(A, b, x)
            x_partialGradient = copy.deepcopy(x)
            x_partialGradient[index] = x_totalGradient[index]
            x = copy.deepcopy(x_partialGradient)
            x_iterations.append(x)
            iter += 1

    if iter > maxIters:
        stoppingReason = f"Max iterations ({maxIters}) exceeded"
    else:
        stoppingReason = f"Gradient norm smaller than {epsilon}"
    return (x, x_iterations, stoppingReason)
```

8.2 Coordinate descent algorithm convergence rate analysis

[61]:
```
# There are three different scales: small, large and huge
scales = ["small", "large", "huge"]
# Number of maximum iterations of the three scales small, large and huge
maxIters = [20000, 40000, 60000]
# Tolerance of the gradient norm of the three scales small, large and huge
tolerances = [10e-5, 10e-3, 10e-1]
# The algorithm
algorithmName = "coordinate descent"
algorithm = coordinateDescent
# The logarithm base
logBase = 2
```

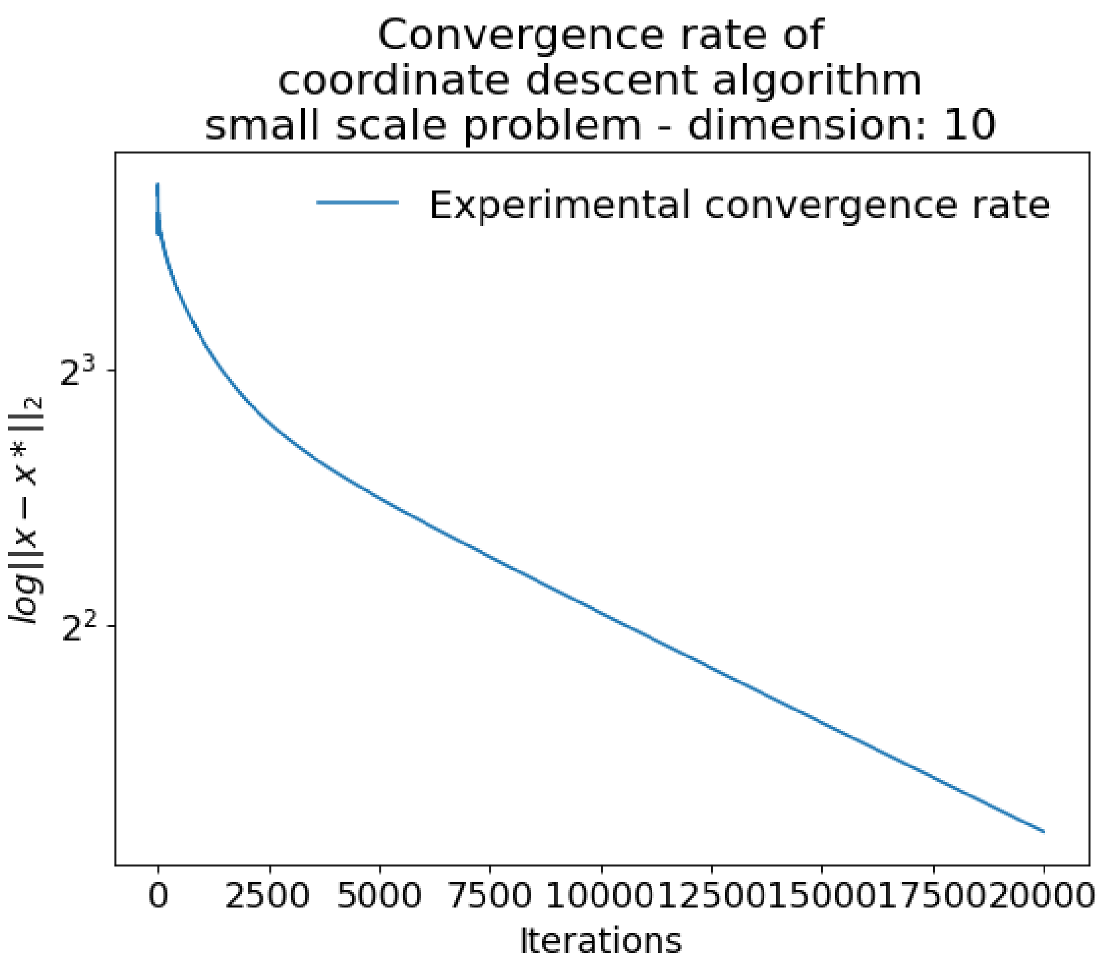
```
# Running optimization for the three scales small, large and huge
for i in range(0,3):
    plotDifferenceNorms(scales[i], maxIters[i], tolerances[i], algorithmName,␣
 ↪algorithm, logBase)
```

```
The small scale problem is chosen. The matrix A and vector b dimension is 10
The number of maximum iterations is 20000. The allowed tolerance for gradient
norm is 0.0001

The coordinate descent algorithm runs in 0.1549978256225586 seconds
Reason of stopping
Max iterations (20000) exceeded

The optimal solution x found by coordinate descent algorithm
[1.47699082 2.35781817 3.46787914 4.50929597 5.40624933 5.13076524
 7.5507039  6.31390472 8.56043183 9.65492076]

The theoretical optimal solution x*
[ 1  2  3  4  5  6  7  8  9 10]
```

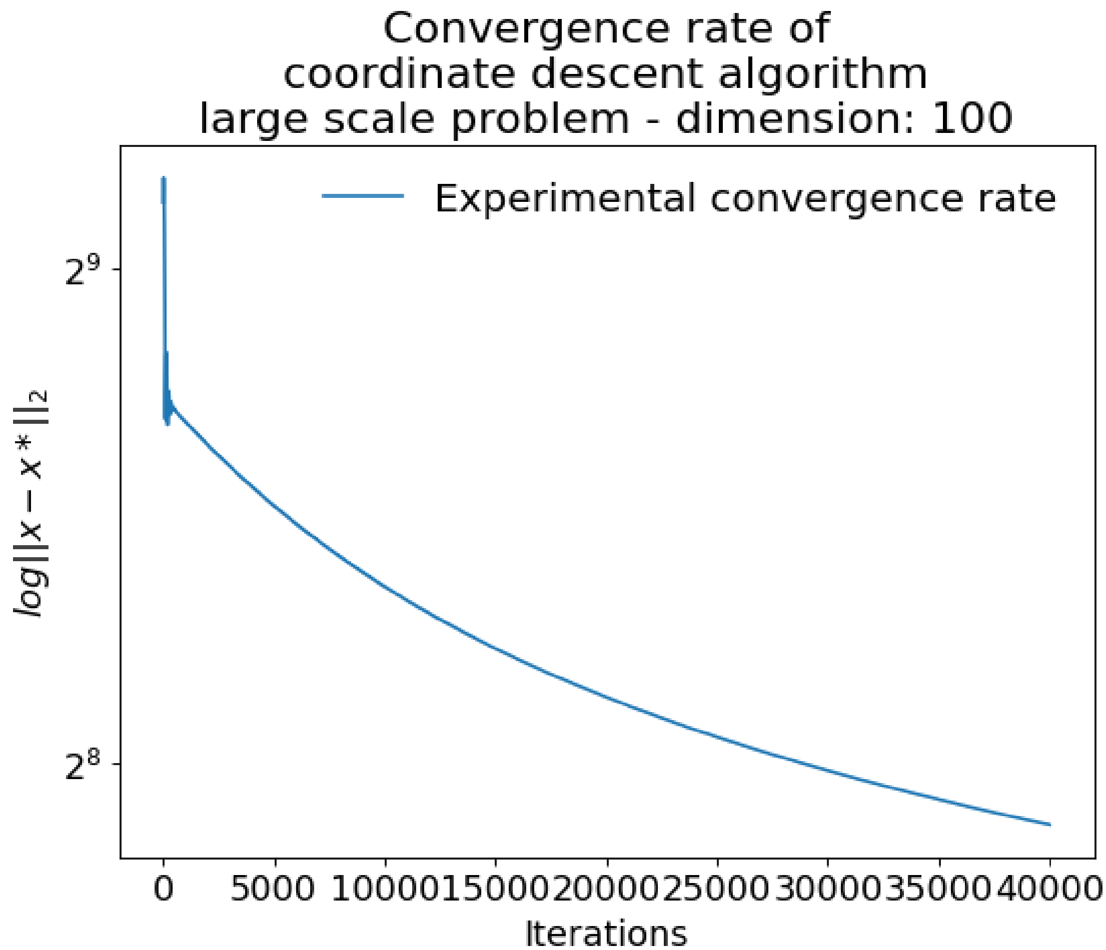


```
The large scale problem is chosen. The matrix A and vector b dimension is 100
The number of maximum iterations is 40000. The allowed tolerance for gradient
norm is 0.01

The coordinate descent algorithm runs in 5.4509971141815186 seconds
Reason of stopping
Max iterations (40000) exceeded

The optimal solution x found by coordinate descent algorithm
[18.4854829  44.25797993 33.84664296 39.61234692 43.26652519 -0.82146242
 15.00451965 52.11998119 10.76369715 25.1453549  28.87342704 37.68531833
 44.19373803 45.315784   24.7130248  39.4887291  39.58141494 57.33326627
 26.06927215 39.95258182 15.22478807  4.79245542 66.18112869 48.69361618
 18.27821879 50.73646993 47.50715397 32.42348842 34.52312645 50.71650713
 60.63219618 42.21262062 41.34583668 45.13040413 25.36149809 65.30948681
 75.59908436 82.54055454 85.61344422 67.17576163 39.51059327 36.51296661
 45.63694754 41.39579049 54.17594465 39.74886786 47.9720558  71.45948733
 62.09519968 44.51460762 39.8757539  51.34473849 57.75974209 53.2257247
 49.44691577 20.17204556 44.4482648  50.61497979 68.89810189 45.26713231
 39.99975888 28.42594745 60.15426635 45.38487869 41.32665787 55.27151341
 62.98501958 53.96928178 75.53946547 65.40910437 73.01644848 68.73756688
 45.00888811 61.54728233 95.44033156 57.37198932 57.90282469 44.17543557
 52.0856743  58.60408173 66.043381   69.46903856 81.10761356 82.18584216
 41.08230713 65.37505301 53.73202672 66.5458011  41.94942278 77.84410287
 66.05262032 41.59800608 54.61866748 58.0782905  76.20893878 48.2623992
 75.94760826 65.24842958 85.82354167 67.84013826]

The theoretical optimal solution x*
[  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
  91  92  93  94  95  96  97  98  99 100]
```



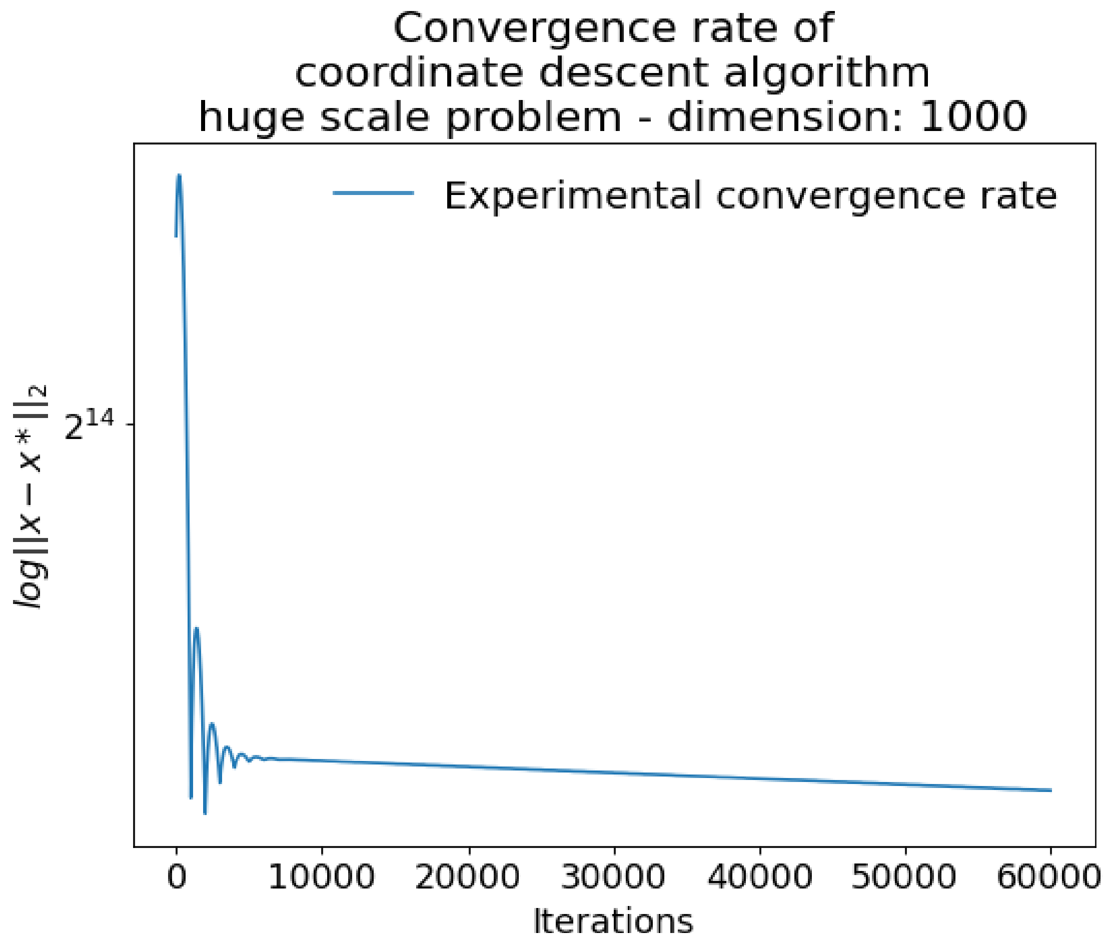

```
The huge scale problem is chosen. The matrix A and vector b dimension is 1000
The number of maximum iterations is 60000. The allowed tolerance for gradient
norm is 1.0

The coordinate descent algorithm runs in 20.09000062942505 seconds
Reason of stopping
Max iterations (60000) exceeded

First 100 values in the optimal solution x found by coordinate descent algorithm
[786.36500442 759.42701392 783.74774341 744.80662387 753.08825931
 742.78808719 760.18211188 777.27424516 757.91373402 758.12745557
 766.10996105 750.37865797 766.10471747 764.91040497 760.78579612
 758.11719986 747.67318075 749.62924301 746.1362748  750.95497023
 752.17386221 778.96218249 747.4290864  714.79019894 758.97535057
 747.10343406 727.02081922 750.73859498 750.97628318 738.76970634
 736.07352565 764.37012303 739.62066681 739.31938756 739.60957661
```

```
 749.5229503  739.1625242  737.6346086  715.51621968 753.15318817
 747.93541246 744.19403141 727.84812933 723.57128145 747.46835544
 765.6173821  741.40360752 762.00495038 756.29983561 719.92925462
 755.82107351 713.52233949 734.80493195 724.90254274 739.17386901
 716.23402562 728.28426787 744.69790816 741.56694161 730.99598664
 730.83657932 718.91793017 718.70380229 708.53268377 723.53976133
 723.48207872 707.61470858 717.52670474 721.07172183 722.39170492
 711.9685472  723.31235392 727.61018016 726.06507322 708.27536654
 734.64569696 699.04454468 708.1519943  720.78399487 691.34651768
 720.93266727 709.69029447 687.15949397 738.30665392 712.75056441
 710.18703696 705.38177504 705.07495572 723.14854648 714.25198636
 727.63531076 702.062696   730.23588956 674.97107686 679.81614971
 707.92066626 703.61102605 687.32613191 710.72097096 705.03773584]

First 100 values in the theoretical optimal solution x*
[  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
  91  92  93  94  95  96  97  98  99 100]
```



From the three graphs, it appears that the implemented coordinate descent has a sublinear convergence rate.

For the small scale problem, coordinate descent returns optimal solution close to x*

For the large scale problem, coordinate descent returns suboptimal solution that has the same pattern as x*

For the huge scale problem, coordinate descent returns suboptimal solution that is still far from x*, again because the dimension is too large (1000)

Coordinate descent error graphs seem to resemble a wavelength gradually flattening out from the earliest iterations until later iterations

9 Task 5: Comparison between the algorithms

[64]:
```
# Plotting the difference norms log ||x - x*||2
def plotDifferenceNormsMutipleAlgorithms(scale, maxIter, tolerance,␣
↪algorithmNames, algorithms, logBase):

    A = np.load(f"data/{scale}Matrix.npy", allow_pickle=True)
    # print("The matrix A")
    # print(A)

    b = np.load(f"data/{scale}Vector.npy", allow_pickle=True)
    # print("\nThe vector b")
    # print(b)
    x_opt = np.load(f"data/{scale}Solution.npy", allow_pickle=True)

    print(f"\nThe {scale} scale problem is chosen. The matrix A and vector b␣
↪dimension is {b.size}")
    print(f"The number of maximum iterations is {maxIter}. The allowed␣
↪tolerance for gradient norm is {tolerance}" )

    if scale == "huge":
        print("\nFirst 100 values in the theoretical optimal solution x*")
        print(x_opt[0:100])
    else:
        print("\nThe theoretical optimal solution x*")
        print(x_opt)

    figure(figsize=(8, 6), dpi=80)

    for i in range(0, len(algorithms)):
        start = time.time()
        x_opt_algo, x_iterations_algo, stoppingReason = algorithms[i](A, b,␣
↪maxIter, tolerance)
        end = time.time()

        differenceNorms = []
        for x_sol in x_iterations_algo:
            differenceNorms.append(differenceNorm(x_sol, x_opt))
        differenceNorms = np.array(differenceNorms)

        iterations = np.arange(0, differenceNorms.size, 1)
        plt.plot(iterations, differenceNorms, label = algorithmNames[i])#,␣
↪marker='.', markersize=5)

    size = 16
```

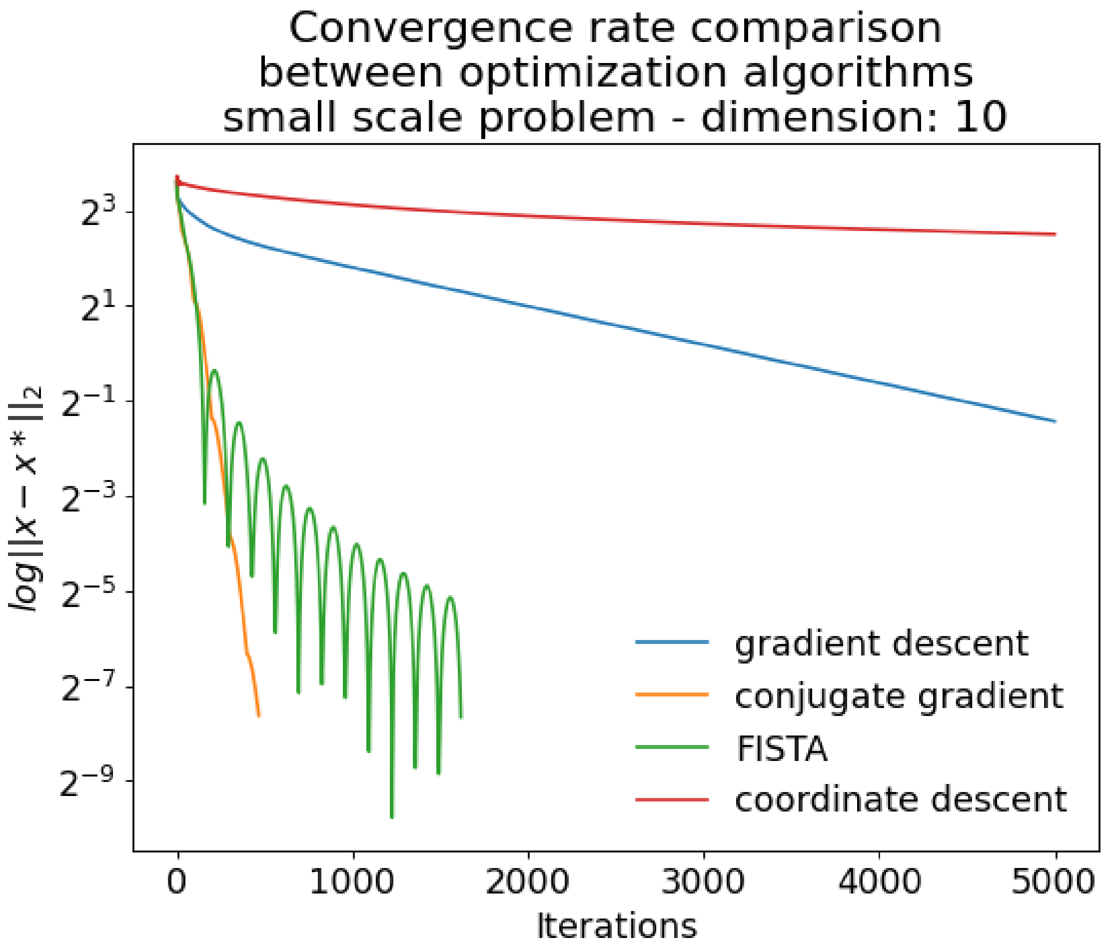
```
    plt.title(f"Convergence rate comparison\nbetween optimization␣
↪algorithms\n{scale} scale problem - dimension: {b.size}", size=size + 4)
    plt.xticks(fontsize=size)
    plt.yticks(fontsize=size)
    plt.yscale('log',base=logBase)
    plt.xlabel("Iterations", size=size)
    plt.ylabel(r'$log||x-x*||_2$', size=size)

    plt.legend(loc=4, frameon=False, fontsize=size, ncol=1)
    plt.show()
```

[65]:
```
# There are three different scales: small, large and huge
scales = ["small", "large", "huge"]
# Number of maximum iterations of the three scales small, large and huge
maxIters = [5000, 10000, 20000]
# Tolerance of the gradient norm of the three scales small, large and huge
tolerances = [10e-5, 10e-3, 10e-1]
# The algorithm
algorithmNames = ["gradient descent", "conjugate gradient", "FISTA",␣
↪"coordinate descent"]
algorithms = [gradientDescent, conjugateGradient, FISTA, coordinateDescent]
# The logarithm base
logBase = 2
for i in range(0,3):
    plotDifferenceNormsMutipleAlgorithms(scales[i], maxIters[i], tolerances[i],␣
↪algorithmNames, algorithms, logBase)
```

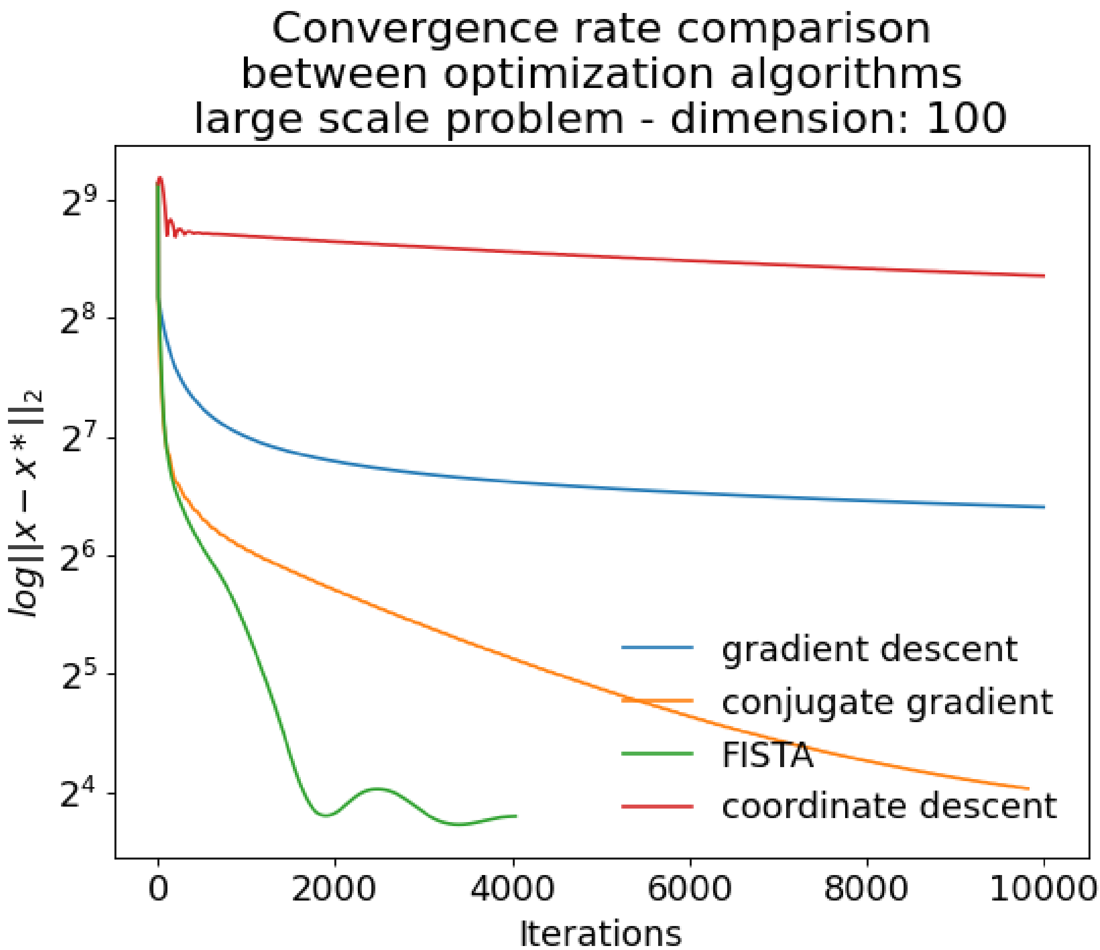
```
The small scale problem is chosen. The matrix A and vector b dimension is 10
The number of maximum iterations is 5000. The allowed tolerance for gradient
norm is 0.0001

The theoretical optimal solution x*
[ 1  2  3  4  5  6  7  8  9 10]
```



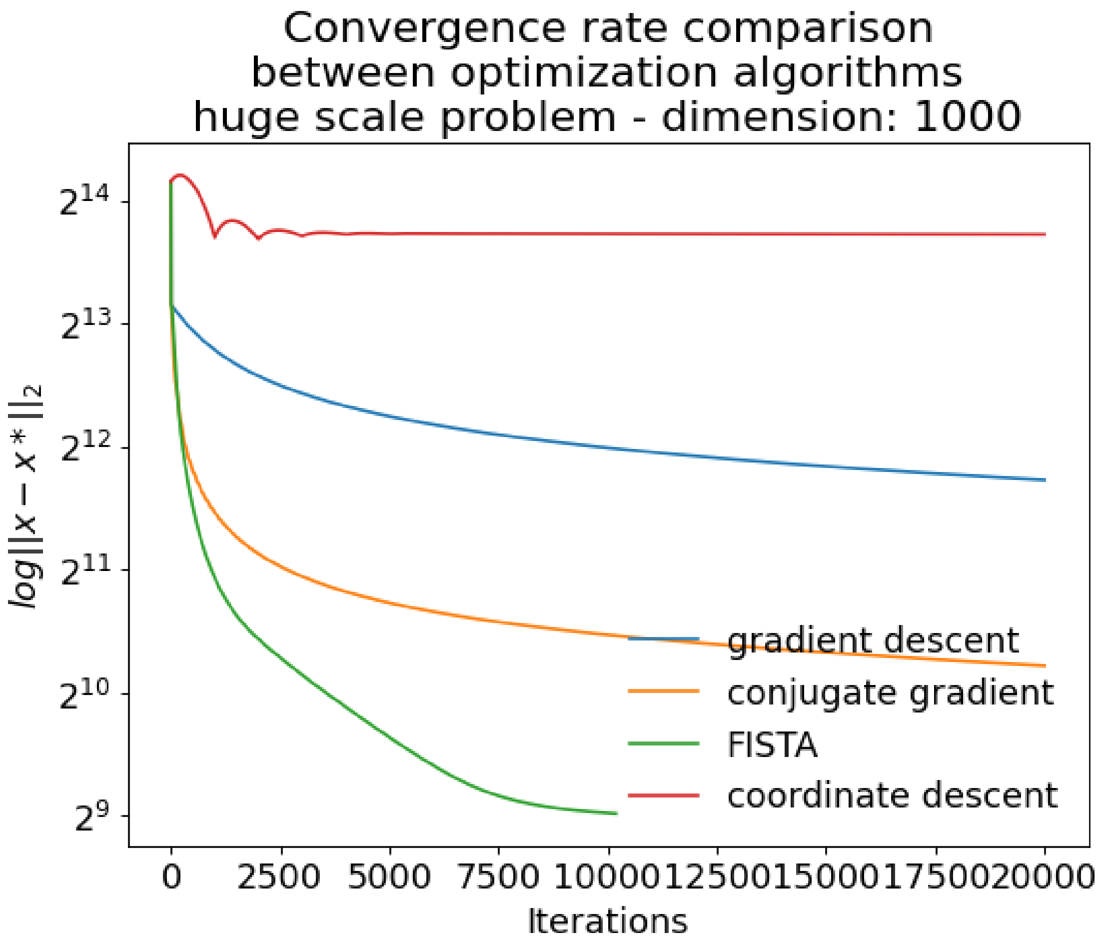

```
The large scale problem is chosen. The matrix A and vector b dimension is 100
The number of maximum iterations is 10000. The allowed tolerance for gradient
norm is 0.01

The theoretical optimal solution x*
[  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
  91  92  93  94  95  96  97  98  99 100]
```




```
The huge scale problem is chosen. The matrix A and vector b dimension is 1000
The number of maximum iterations is 20000. The allowed tolerance for gradient
norm is 1.0

First 100 values in the theoretical optimal solution x*
[  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
  91  92  93  94  95  96  97  98  99 100]
```



From the three comparison graphs, we can finally conclude the performance of each algorithms

1. Gradient descent: Normal linear convergence rate in small dimension and slightly sublinear convergence rate in higher dimensions
2. Conjugate gradient: Fast superlinear convergence rate in all dimensions
3. FISTA: extremely fast superlinear convergence rate in all dimensions
4. Coordinate descent: Slow sublinear convergence rate in all dimensions

The speed of convergence rankings are therefore:

- Small dimension (10): conjugate gradient > FISTA > gradient descent > coordinate descent
- Large dimension (100): FISTA > conjugate gradient > gradient descent > coordinate descent
- Huge dimension (1000): FISTA > conjugate gradient > gradient descent > coordinate descent

Gradient descent is popular in many ML algorithms and solvers in deep learning. Particularly, stochastic gradient descent is much more useful in batches training, where updating the training performance with the whole data is expensive or impossible.

Conjugate gradient is applicable to sparse systems that are too large to be handled by a direct implementation or other direct methods such as the Cholesky decomposition

FISTA is the fastest algorithm and is robust against large dimensions, making it highly suitable for solving many optimization problems involving a large number of parameters.

Coordinate descent should be used for problems where individual updates are much easier than the whole updates of all components, such as LASSO method in ML. Therefore, coordinate descent is useful in distributed optimization problem.

Visually, gradient descent is going in straight line in a Euclidean map towards the optimum, while coordinate descent follows along only one variable at a time like a stair case, which means it traverse the Manhattan distance towards the optimum. As a result, the gradient descent strictly converges faster than coordinate descent because straight line distance is always larger than the Manhattan distance.

Conclusion: We should use FISTA for large scale problems and conjugate gradient for small scale problems, if applicable, thanks to their fast convergence speed. If not, (stochastic) gradient descent is highly recommended, as it has been implemented for many existing problems. If individual updating is much easier than total updates, coordinate descent is the most suitable algorithm