ANNOUNCEMENT

Mid-term Exam

- Tentative date: **Mar 29 Tuesday** (after Spring break).

  --2-3:20pm; same time as class.

  --online exam: download before 2pm; submit to gradescope. (details on wiki page "Mid-term exam information")

- Send me an email if you are not available!

- Open to lecture slides, textbook, homework. Not allow to check any other material.
- No communication with anyone, except TA and instructor.

  --any sign of communication may cause in penalty.

Projects

- **Topic proposal** is due in a few weeks: Mar 12

- Check wiki page "Project suggestions and timeline"
  --timeline
  --listed a few candidate topics (other topics are fine)
  --other instructions

Also check "Lecture 0" on the project.

Ask questions on Piazza. **Act early. Act early. Act early.**

CG METHOD AND COMPARING ALGORITHMS

Ruoyu Sun

Learning Goals of This Lecture

Conjugate gradient method (brief intro) and compare earlier algorithms

- **After today's lecture, you should be able to**
  - Tell why Conjugate Gradient method is popular for quadratic problems
  - Compare algorithms using synthetic data
  - Tell a few tricks and traps of the comparison

PART I
CONJUGATE GRADIENT METHOD

CG: Method of Choice for Quadratic Problems

- **Conjugate gradient method**: originally designed for solving symmetric PD **linear system Qx = c**

  --same as solving min $x'Qx - 2c'x$

  often the method of choice; well known for decades

Remark: For non-symmetric linear system, other methods are popular.

- Extendable to nonlinear problems (**non-linear CG**).

CG: History

- *Linear* **conjugate gradient (CG)** method was proposed by Hestenes and Stiefel in the 1950s;

  --as an iterative method for solving PD linear systems

- First *nonlinear* CG method: Fletcher and Reeves, 1960s.

  --among earliest techniques for large-scale nonlinear optimization problems (before BFGS)

Newton-CG in Scipy

- Scipy.optimize provides “CG” and “Newton-CG”
- https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html

- Method Newton-CG uses a Newton-CG algorithm (also known as the truncated Newton method); see [Numerical optimization] pp. 168

- “It uses a CG method to the compute the search direction”

- In Newton method, the search direction is $\mathrm{v} = H^{-1}g$, where $H = \nabla^2 f(x^k)$, $\mathrm{g} = \nabla f(x)$.
- Computing $\mathrm{v} = H^{-1}g \Leftrightarrow$ solving $\mathrm{Hv} = g$
- (i.e., replace matrix inversion by solving linear system with CG method)

- What if Hessian is not PD?
  --Start from $\mathrm{v}(0) = -\nabla f(x)$;
  --update $\mathrm{v}(k)$ until convergence, or getting non-descent direction

(nonlinear) CG in scipy

Scipy.optimize provides “CG” and “Newton-CG”

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

- Method CG uses nonlinear CG by Polak and Ribiere
  --variant of the Fletcher-Reeves method
  --see [Numerical optimization] pp. 121-123.

- We will briefly discuss Fletcher-Reeves method later.

Linear Conjugate Gradient Method

Solving symmetric PD linear system Ax = b, or quadratic problem min x'Ax/2 – b'x.

Given x_0;
Set $r_0 \leftarrow Ax_0 - b$, $p_0 \leftarrow -r_0$, $k \leftarrow 0$;
while $r_k \neq 0$

Initialize residual r, direction p

$$\alpha_k \leftarrow \frac{r_k^T r_k}{p_k^T A p_k};$$ Stepsize

$x_{k+1} \leftarrow x_k + \alpha_k p_k;$ Update iterate

$r_{k+1} \leftarrow r_k + \alpha_k A p_k;$ Update residual

$$\beta_{k+1} \leftarrow \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k};$$

$p_{k+1} \leftarrow -r_{k+1} + \beta_{k+1} p_k;$ Update direction, by new residual

$k \leftarrow k + 1;$

Non-linear CG Method (Fletcher-Reeves)

Solving min f(x), for general differentiable function f.

Algorithm 5.4 (FR).

Given x_0;

Evaluate $f_0 = f(x_0), \nabla f_0 = \nabla f(x_0)$;

Set $p_0 \leftarrow -\nabla f_0, k \leftarrow 0$; Initial direction is negative gradient.

while $\nabla f_k \neq 0$

Compute α_k and set $x_{k+1} = x_k + \alpha_k p_k$; Update iterate

Evaluate ∇f_{k+1};

$$\beta_{k+1}^{\mathrm{FR}} \leftarrow \frac{\nabla f_{k+1}^T \nabla f_{k+1}}{\nabla f_k^T \nabla f_k}$$; Update "momentum" coefficient

$$p_{k+1} \leftarrow -\nabla f_{k+1} + \beta_{k+1}^{\mathrm{FR}} p_k$$; Update direction; utilizing new grad info

$$k \leftarrow k + 1;$$

Disclaimer: Skipping Details of CG

- The above explanations are **very brief**.
- To fully understand CG method, one shall learn its deeper motivation and a few properties
- We skip these details, but only focus on the importance and theory of CG

Three Main Results on Linear CG

For solving PD linear system Ax = b, CG method is very fast.

Theorem 5.4.

If A has only r distinct eigenvalues, then the CG iteration will terminate at the solution in at most r iterations.

Theorem 5.5.

If A has eigenvalues $\lambda_1 \le \lambda_2 \le \cdots \le \lambda_n$, *we have that* (denote $\| y \|_A^2 = y^T A y$)

$$\|x_{k+1} - x^*\|_A^2 \le \left(\frac{\lambda_{n-k} - \lambda_1}{\lambda_{n-k} + \lambda_1}\right)^2 \|x_0 - x^*\|_A^2.$$

error $\frac{\|x_t - x^*\|_A}{\|x_0 - x^*\|_A}$ decreases like: $1 - \frac{\lambda_1}{\lambda_n}, 1 - \frac{\lambda_1}{\lambda_{n-1}}, \ldots, 1 - \frac{\lambda_1}{\lambda_2}, 1 - \frac{\lambda_1}{\lambda_1} = 0$

Result 3: $$\|x_k - x^*\|_A \le 2\left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1}\right)^k \|x_0 - x^*\|_A.$$

Convergence Rate of Linear CG

- For solving PD linear system Ax = b, CG method is very fast.
- **First**, terminate in at most n iterations.
- **Second**, "**multi-stage" behavior**: at the t-th iteration, the error is dependent on the ratio of the t-th largest eigenvalue over min-eigenvalue

  -most important reason why CG is fast
- **Third**: at least linear convergence at rate $1 - 1/\sqrt{\kappa}$

  --this rate **bound:** same as NAG for convex quadratic problems

  --the actual convergence of CG is much faster than NAG

Performance of CG: first few iterations

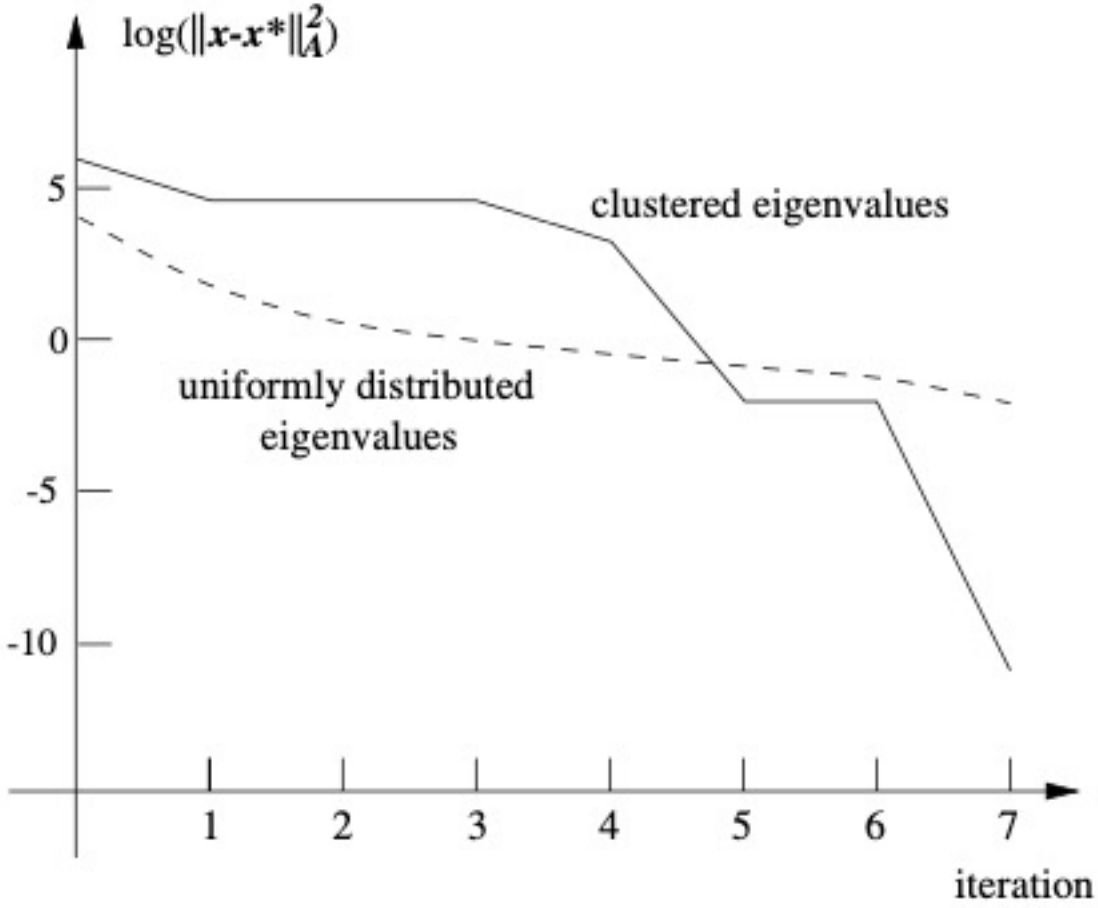


Figure 5.5 of [Numerical optimization]

Relation of BFGS and CG

- When using exact line search for solving strongly convex quadratic problems, BFGS is equivalent to (linear)-CG
  (see discussion in Bertsekas “nonlinear programming”).

- Partially explain excellent performance of BFGS
  --**Remark:** BFGS has multiple steps of approximation, so theoretical benefit less clear without this result

- For general **non-quadratic** problems, or BFGS with inexact linear search, BFGS and (nonlinear)-CG are different.

PART II EXPERIMENTS ON DIAGONAL MATRICES

Methods, Theory and Experiments

- **Methods:** we discussed various methods: GD, HB & NAG, BB, BFGS, CG

- **Theory**: we have a bunch of **theoretical results**
  --convergence rate results
  --discussion of their speed

- **Experiments**: we have a few experimental studies
  --1D logistic regression
  --multi-stage behavior in low-dim linear regression

What is missing: comparing these methods in high-dim experiments

Comparing Methods Is Tricky

- **Comparing methods is tricky!**
- **Comparing methods is tricky!!**
- **Comparing methods is tricky!!!**

- **Common "mistake":**
  --generate "random" data
  --compare two methods and draw conclusion

  Your conclusion: Algo-1 is better than Algo-2
  What you really did:
  --on a linear regression problem with Gaussian data
  --with zero-ini point and constant stepsize 0.1
  --Algo-1 achieves smaller iterate error than Algo-2 within 5000 iterations

Ideal Comparison

- **Comparing two methods require:**

- **Diverse problems and instances (data).**
  --**problems**: from many application areas, different forms (e.g. Mittleman's standard test datasets)
  --**instances**: for a given form (linear regression, or fitting financial data), various data sources

- **Tuning algorithm hyperparameters (for each instance)**
  --initial point
  --stepsize, momentum coeff., etc.
  --max # of iteration; max line search steps; max memory steps, e.g.

- **Choose proper metric**
  --Target accuracy, 1e-5, 1e-8, or 1e-12?
  --gradient error?
  --function value? Or function error?
  --iterate difference? Or iterate error?

Tip: How to memorize these?
If you prepare data, write algorithm and make plot yourself, then every part can be tunable.

This Lecture

- **This lecture, we compare BB, BFGS, GD, etc. For simplicity, we explore:**

- **Diverse problems and instances (data).**
  ~~--**problems**: from many application areas, different forms (e.g. Mittleman's standard test datasets)~~
  --**instances**: for a given form (**linear regression**), various data sources

- **Tuning algorithm hyperparameters (for each instance)**
  ~~--initial point~~ (one random initial point)
  ~~--stepsize, momentum coeff., etc.~~ (fixed hyperparams)
  --max # of iteration; ~~max line search steps~~; ~~max memory steps~~, e.g.

- **Choose proper metric**
  --Target accuracy, 1e-5, 1e-8, or 1e-12?
  --gradient error?
  ~~--function value? Or function error?~~
  ~~--iterate difference? Or iterate error?~~

Setting 1: Diagonal Matrices

- **The problem in the whole lecture:**

  min x'Qx.

- **Toy Case 1.0**: Q = (1, 1e-2, 1e-4).

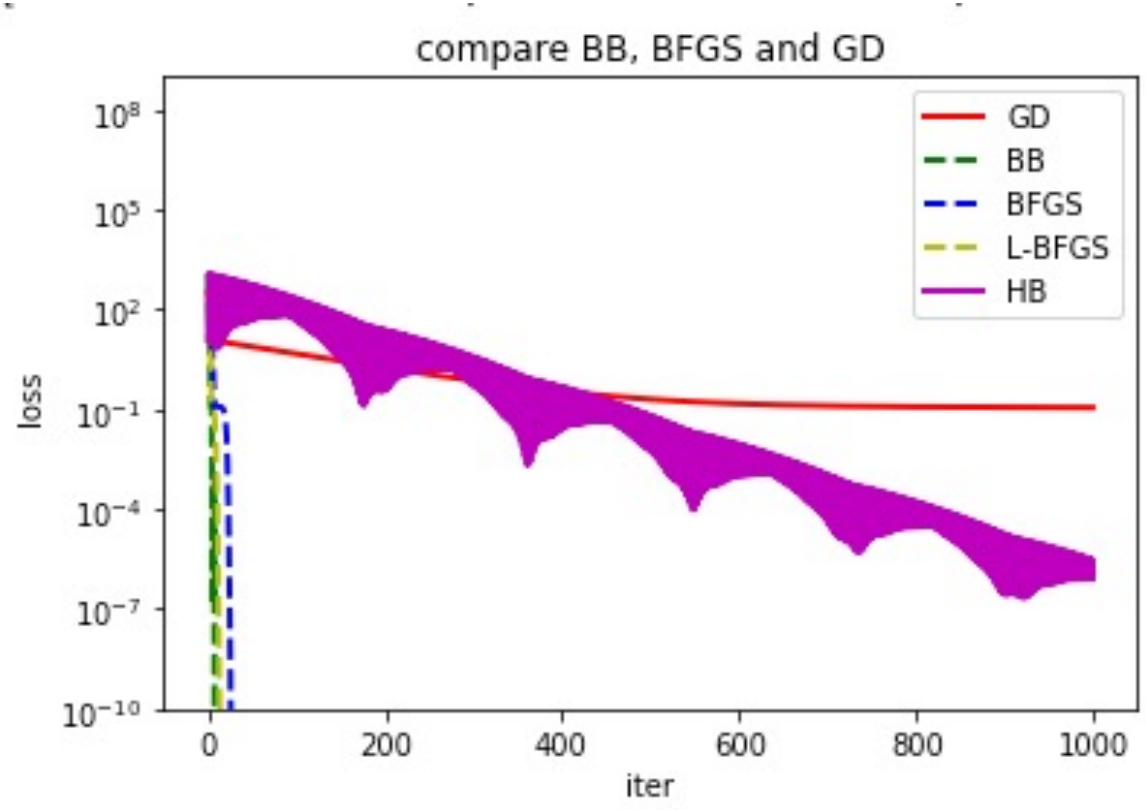
- We discussed this toy case before.

Case 1.1: Three Clusters

- Change the eigenvalue distribution.
- Case 1.1: **Spectrum consists of** three clusters.

D1 = (1, 1, …, 1, 1e-2, …, 1e-2, 1e-4, …, 1e-4)
Q = D1

d=100, Spectrum: (1, 1, …, 1, 1e-2, …, 1e-2, 1e-4, …, 1e-4)



With only three clusters,
BB, BFGS, L-BFGS are 100+ times faster.

Case 1.2: Linear Spacing

Case 1.2: **linear spacing** between eigenvalues.

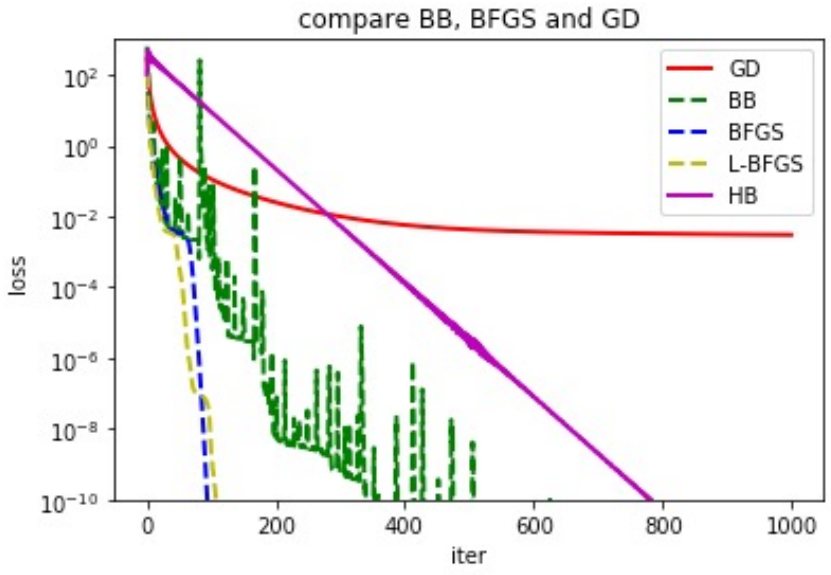
- d = 100
- spacing = 30
- v = np.arange(1, d*spacing, spacing)
- D2 = np.diag(v/max(v))
- Q = D2

Eigenvalues are: 1, 31, 61, 91, 121, 151, …, 2971

--scale all eigenvalues so that the maximal eigenvalue is 1;
so minimum eigenvalue is 1/2971 ≈ 0.00034.

Case 1.2: **linear spacing** between eigenvalues

```
L =  1.0
mu = 0.0003365870077415012
```



HB is much faster than GD.

BB is 2-3 times faster than HB.

BFGS, L-BFGS take fewer iterations than BB (total cost maybe comparable?)
--gaps increase as error increases

Tuning Data generation

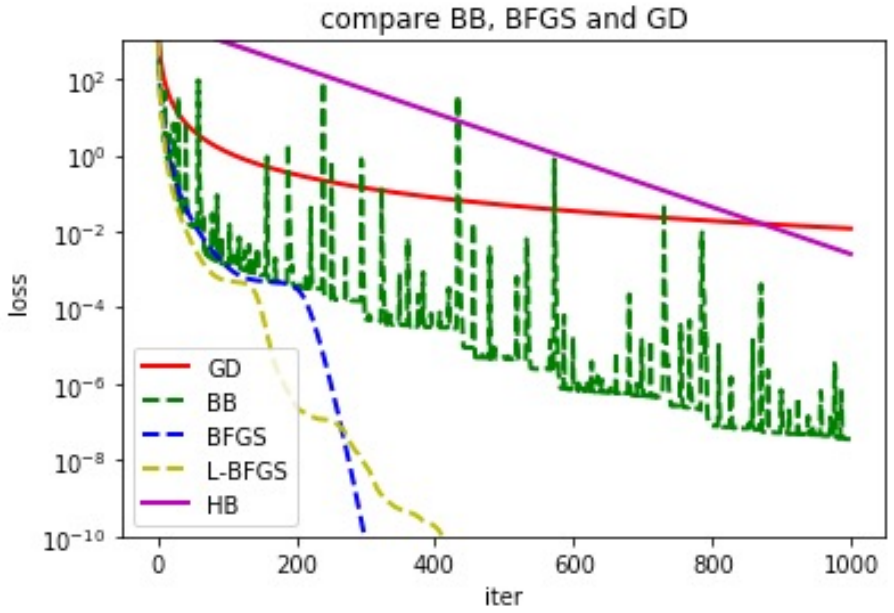
- What if spacing = 1?
- --Performance of HB should be good

- What if using different initial point?

What if using different max-iteration?

```
d = 1000
spacing = 20
```

```
L =  1.0
mu = 5.004754516790951e-05
```



Gap between BB and HB is even larger

GD is faster than HB (in 800 iterations).
--Higher dim often requires more iterations; o.w. conclusion can be reversed

BFGS outperforms BB after getting 1e-3 error.

Generic Advice of Looking at Figures

How does tuning parameter will affect the performance?

--Especially eigenvalues.

It is good to keep in mind that:

1) High accuracy v.s. low accuracy;

2) What if this scales to high-dim?

3) Shall we look at first 100 iterations? What if run 10 times more?

PART III RANDOM MATRICES

Random matrices

- In practice, often test the algorithms using two kinds of data:

- 1) **Synthetic data.**

  --“Random” data ;

  --data with controlled spectrum.

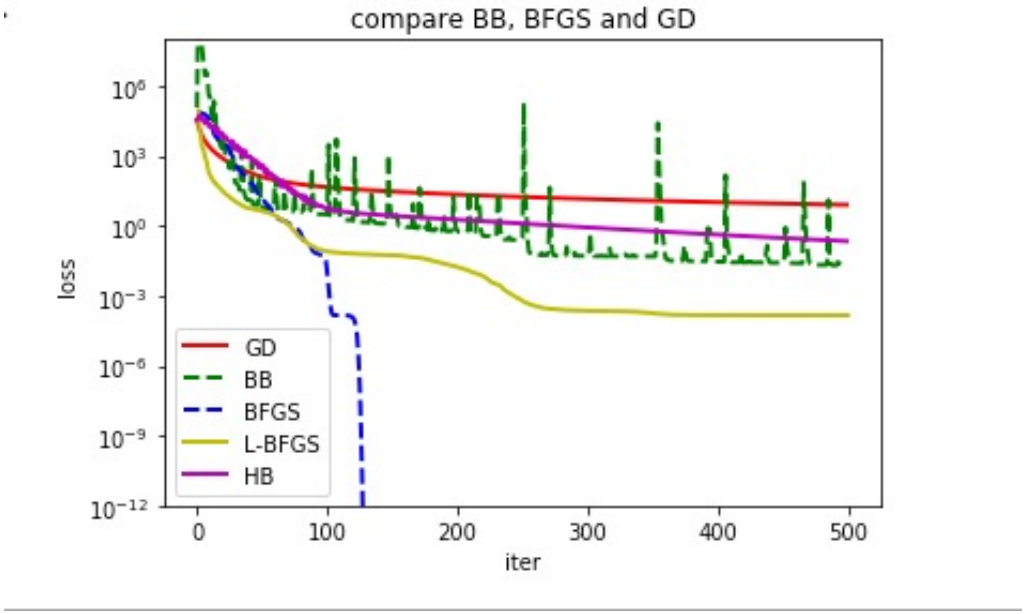
- 2) **Real data. –--Skip today**

Setting 2.1a: Random Gaussian Data

d = n = 100

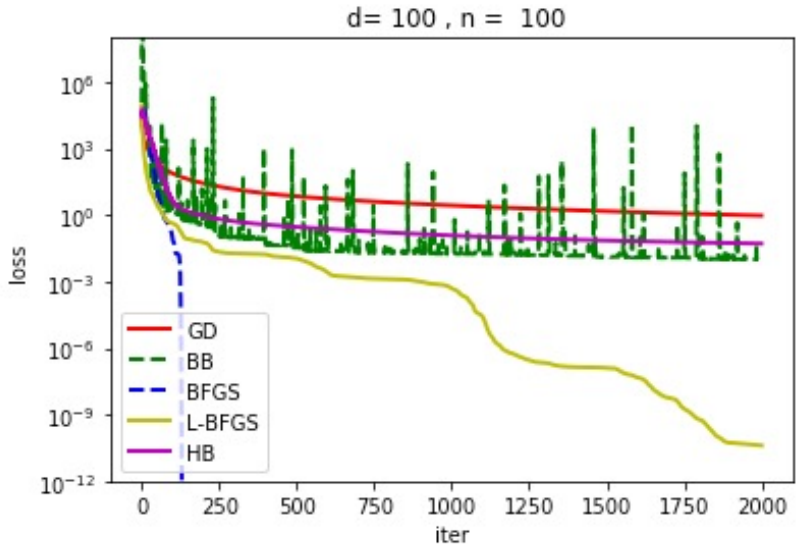
A = np.random.randn(n,d)

Q = A.T @ A



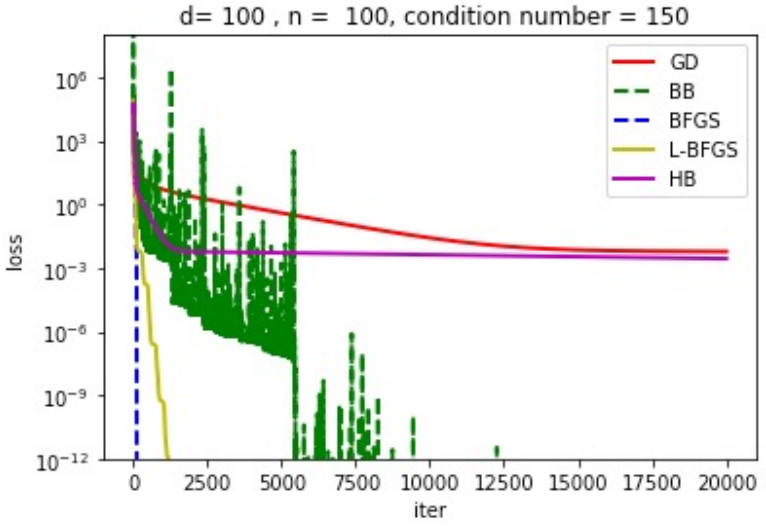
Lessons?

- Except BFGS, all other 4 methods are similar?
- Remember the tip: run it for longer.



2k iterations

BB looks similar to HB



20k iterations

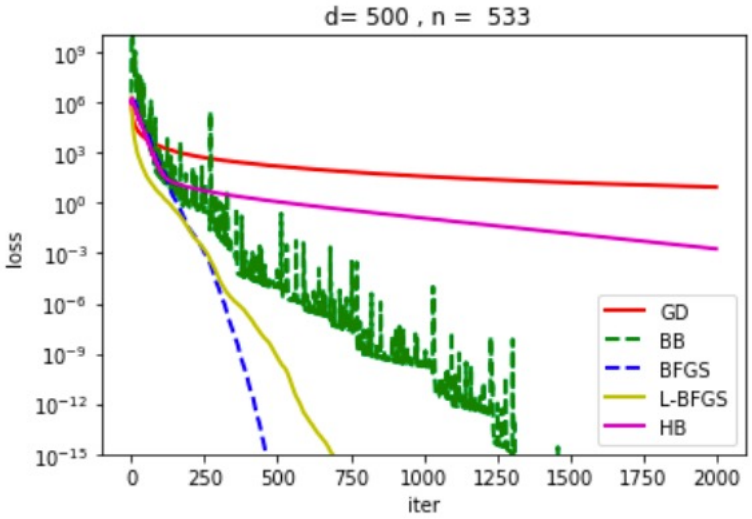
BB is much faster than HB.

Change: unequal n and d

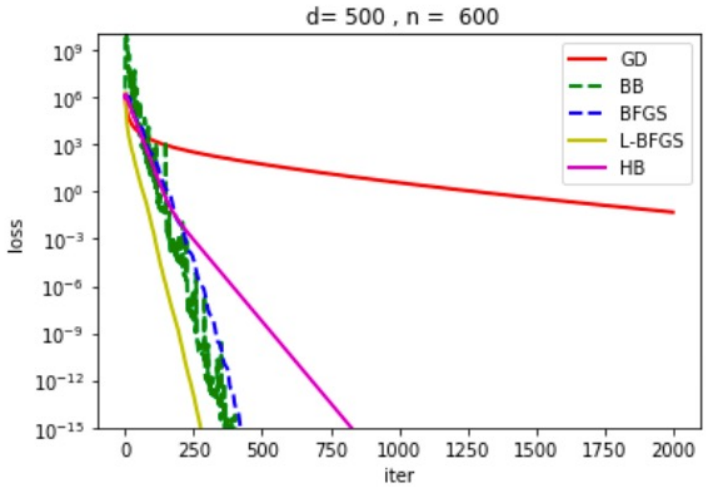
- We assumed n = d for simplicity.
- In data analysis, # of samples often is larger than # of features, i.e., n > d.
- What if we use n > d, instead of n = d?

Setting 2.1b Rectangular, d = 500

- **d = 500, n = 533. 2000 iterations**
- L = 2076, mu = 0.42. $\kappa \approx 5000$



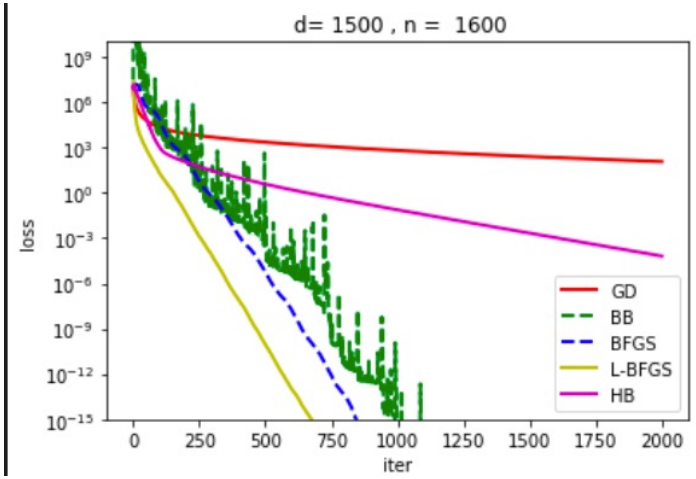
d = 500, n = 600. 500 iterations
L=6583, mu = 15,$\kappa \approx 400$



Gap between BB and HB: increasing as κ increases from 400 to 5000

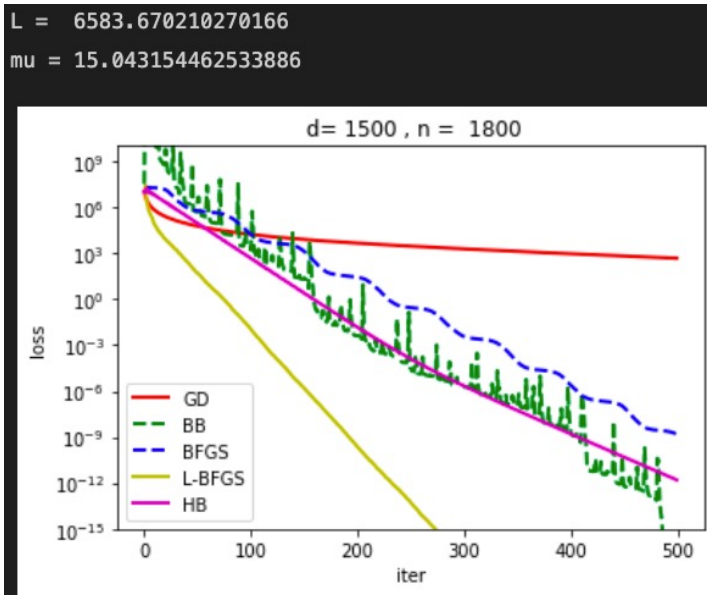
Setting 2.1b Rectangular, d = 1500

- **d = 1500, n = 1600**.
- L = 6244, mu = 2.02. $\kappa \approx 3100$



2000 iterations

d = 1500, n = 1800.
L=6583, mu = 15,$\kappa \approx 400$



500 iterations

Gap between BB and HB: increasing as κ increases from 400 to 3000

Ratio d/n matters more than size of d: (d,n)=(1500,1800) similar to (500,600)

Math knowledge: Why unequal n and d make so huge difference?

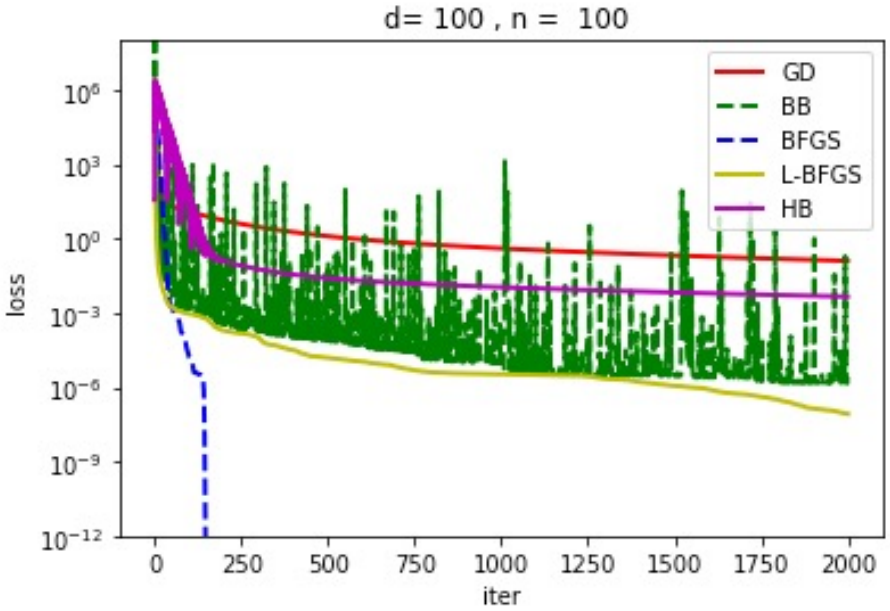
- Need a bit background of random matrix.
- Edelmen: Eigenvalues and condition numbers of **random matrices** (Edelmen has many papers on this)
- Pay attention: Q = A'*A, where A is random, is called Wishart matrix.

Result 1: same n/d ➔ similar eigenvalue pattern ➔ similar convergence comparison

- **Result 2**: for square Gaussian A, kappa **is huge**; for non-square Gaussian A, **kappa is much nicer**.
- **Exercise**: plot eigenvalues of square and non-square A

Setting 2.2a: Uniform random data

- A = np.random.rand(n, d)
- Q = A.T @ A



Observation: BB and BFGS are similar (lower envelop); faster than HB.

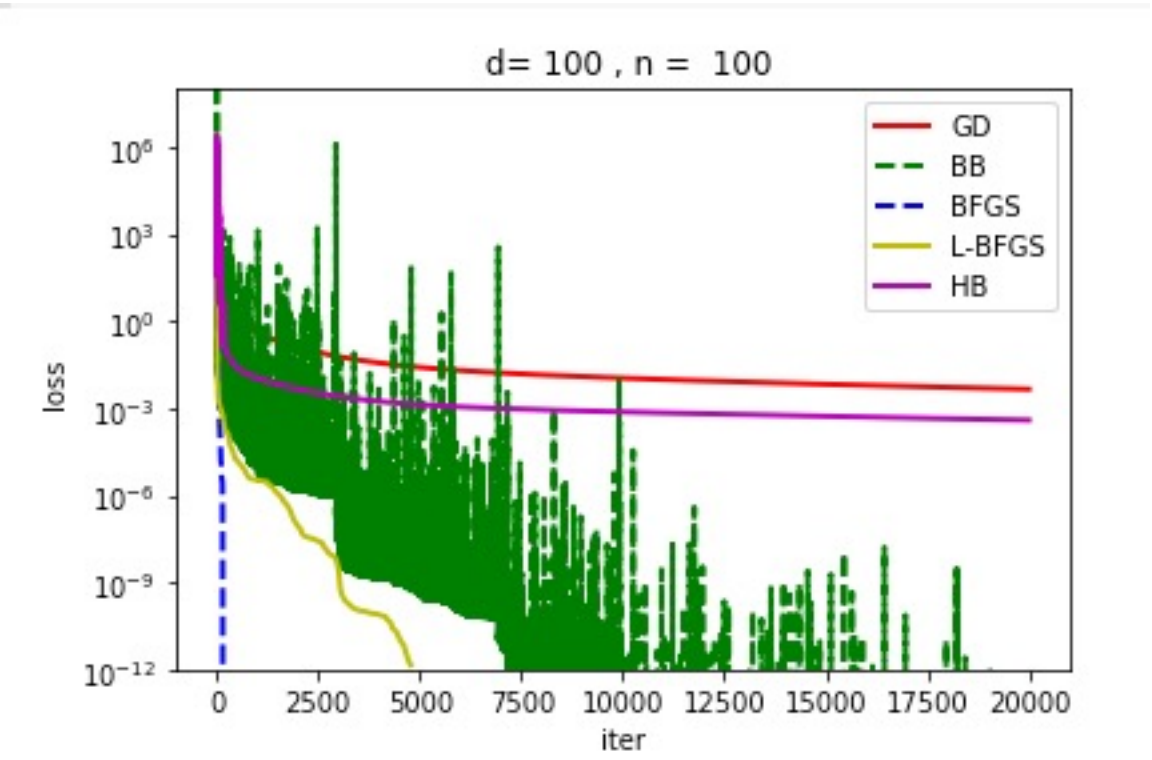
What's next interesting experiment?

Longer training?

20k iterations.

BB is so much faster than HB and GD (why? Check condition number)

L-BFGS has not shown too much benefit over BB.

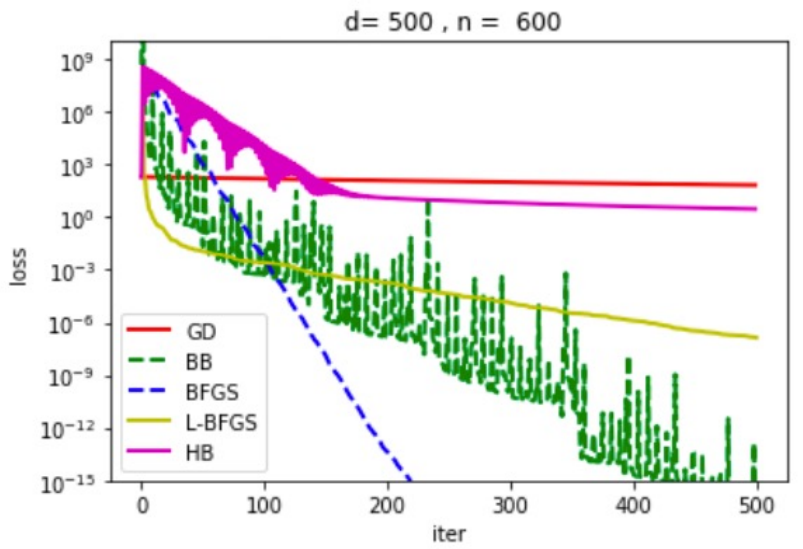


Setting 2.2b Rectangular Uniform

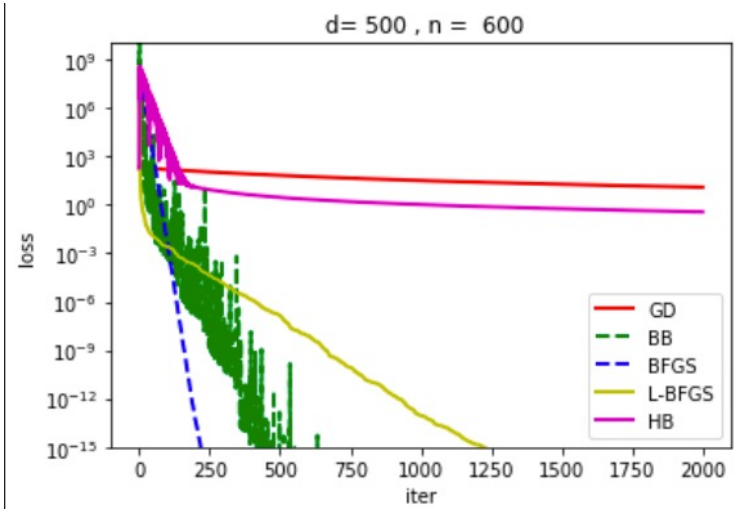
d = 500, n = 600. Uniform random.

GD and HB are slow; BB is really good.
BB takes fewer iterations than L-BFGS!!

```
L =  75043.13128736112
mu = 0.41551281135909196
```



2000 iterations



Lesson on BB v.s. L-BFGS

- **Only case** so far: BB takes fewer iterations than L-BFGS.

  --Considering the extra cost of L-BFGS, BB is clearly dominating L-BFGS in this case.

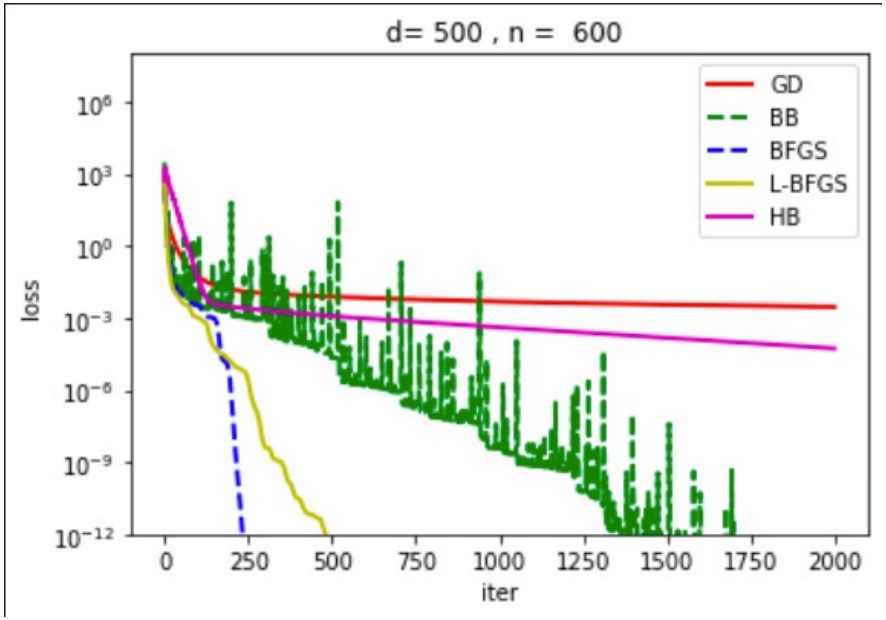
- After seeing 10+ settings, we thought: *L-BFGS always takes fewer iterations than BB*

  --too early to make assertion.

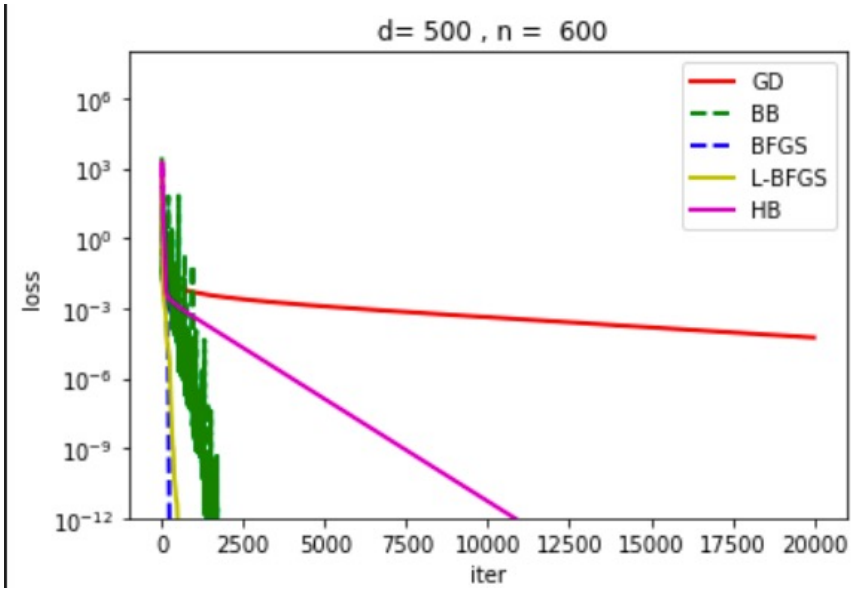
Setting 2.3: Controlled Condition Number

- Random eigenvalues with $\kappa = 1e5$ (set D = D3)
- Set A = A3, i.e.,

  A = UDV', where U, V are orthogonal matrices (random Gaussian matrices).



2k iterations



20k iterations

BB is much faster than HB (about 5-6 times faster)
BFGS, L-BFGS takes fewer iterations than BB.

Summarize These Settings?

"<" means "takes fewer iterations"

- **Low accuracy:**
  random data BFGS ≈ L-BFGS **<** BB < HB < GD
  (control-kappa case: L-BFGS << BB at 1e-6)

  High accuracy:
  BFGS < L-BFGS **<<** BB **<<<** HB < GD

  Here, we ignored per-iteration time!

(so L-BFGS < BB does not mean L-BFGS is faster than BB in terms of CPU time)

Disclaimer

- This lecture focused on "# of iterates" comparison
- This is NOT "**time" comparison**
- **You shall report CPU time comparison between algorithms (in projects)**, especially if they are from different classes
  --e.g. compare BB and BFGS
  --it's fine if they have similar per-iteration cost (e.g. HB v.s. NAG v.s. BB)
- Issue of CPU time comparison:
  --depend on software (C, python, matlab, etc.)
  --depend on hardware (Macbook, HP, GPU, etc.)
  --depend on implementation (BFGS's different versions)

Other Factors for Experiments

- What if we apply **data processing**? (normalization and centralization)
- **Do eigen-vectors make a difference**? (i.e. does a diagonal matrix and a non-diagonal matrix with the same eigenvalues lead to the same performance?)
- How do parameters of BFGS and L-BFGS like **maxcor and maxls affect the performance**?
- How does momentum parameter beta in HB method affect the performance?

Comparison on Logistic Regression

- Check
- http://fa.bianp.net/blog/2013/numerical-optimizers-for-logistic-regression/

- It compares various methods, including BFGS, L-BFGS, CG and trust region methods.

Suggestions of Experiment Settings

Suggested experiments on comparing algorithms
(either on a real problem, or just comparing algorithms):

0) Real data (one instance, or collection of problems) are the best testbed;

1) If you can plot the eigen-distribution for the problem at hand:
then create a **smaller-size** problem **with similar spectrum pattern**,
and then test the algorithms.

- This is the rational of many papers that study the Hessian spectrum of neural networks.

2) If not, then try two experiments:

- 2.1) Smaller subset of data.

2.2) "Few outliers + small cluster" spectrum (D3 matrix in ipynb file). Possibly modify the small cluster.

Possibly try random U and V (singular vectors).

Remark: similar **(# of iterations)/dimension; similar accuracy.**

3) If try random matrix, use **unequal** d and n. Try both Gaussian and uniform.
(if possible, try data preprocessing)

4) Other factors: sparsity; hardware implementation.

Summary of This Lecture

- **Conjugate gradient (CG) method**:
  --(linear) CG is very popular for solving linear systems
  --nonlinear CG is a possible candidate for nonlinear problems

Theory: multi-stage error; $O(\sqrt{\kappa})$ ite-complexity; n-iteration terminates

- **Experiments on** which methods are good.
  --BB or L-BFGS are good choices
  --BB is bumpy
    L-BFGS requires line search and more memory

- **Second**, need to properly pick test problems to evaluate algorithms.
  --Pay special attention to spectrum
  --Consider changing accuracy, MaxIter, dim ratio, etc.

Comments on This Lecture

- Probably NOT existent on other optimization courses

- **Reason**: traditional optimization courses focus on 2nd order methods (BFGS, DFP, …)
  --HB, NAG, GD are not that important
  --So comparing with HB/NAG is not important

- **Nowadays**, HB/NAG/GD are popular in big data
- You may wonder: which method is better?

- Here we show: BB/L-BFGS are indeed much better
- (Disclaimer: for medium-scale problems)
- HB/GD are popular in huge-scale problems partially due to the necessity of SGD