E0 230 CMO Assignment 2 - 29/09/2025 - 37 points

Prof Chiranjib Bhattacharyya

Instructions

1. Read all instructions before starting the assignment
   Read them again before submitting the assignment.
2. Submissions that do not adhere to the instructions will be given **zero marks**.
3. This is an **individual assignment**, all work including code should be your own.
4. **Do not use AI generated code.**
5. Any form of academic dishonesty will be treated as per the IISc and CSA academic integrity policy.
6. You will submit a **single** `PDF` file, and a **single** `.py` file **and two .txt files for Question 1.2.** for this assignment. Upload the files directly; **do not zip them**.
   (a) Name the `pdf` file **CMO2025A2_vwxyz.pdf** where vwxyz is your five-digit SR number. The pdf file should be generated using the **LaTeX template provided in the Class Materials folder** in the Files Section of the Team. Your pdf should include all values, justifications, and graphs (if any) that you are asked to report.
   (b) Name the `.py` file **CMO2025A2_vwxyz.py** where vwxyz is your five-digit SR number. Your python file should contain all code that you used to solve this assignment. **Only `.py` files are accepted**; no notebooks.
   (c) **Name the .txt files plist_vwxyz.txt and dlist_vwxyz.txt where vwxyz is your 5-digit SR number.**
7. We will not open or evaluate any other file submitted.
8. Incomplete submissions (without the form, `pdf`, or `.py` file) will not evaluated, 0 marks will be awarded.
9. **This assignment is due by 23:59, 10 October.**
10. Late submissions will incur a 20% penalty per day.

Oracle Instructions

1. Use the Oracle in a **Unix-like** terminal.
2. Windows users must run it via Windows Subsystem for Linux (WSL).
3. Run all code for this assignment in a virtual environment with Python version 3.10.
4. Unzip the Oracle into the directory you are working in.
5. Do not modify the `oracle_2025A2_py310` folder.
6. Place your `.py` file and the `oracle_2025A2_py310` folder in the same directory. Read the instructions in the README.md file inside the `oracle_2025A2_py310` folder.
7. **Set up the Oracle by 23:59, 6 October.**
8. No support will be provided for oracle issues after this deadline.

QUESTION 1 (15 points)

1. (1+1+3 points) Let Q be a symmetric positive definite matrix and $f(x) = \frac{1}{2}x^\top Qx - b^\top x$. Suppose we generate directions $\{u_k\}$ using the Gram–Schmidt recursion with the Q-inner product:

$$d_{k+1} = p_{k+1} - \sum_{i=0}^{k} \frac{p_{k+1}^\top Q d_i}{d_i^\top Q d_i}\, d_i,$$

where $\{p_k\}$ are the raw residuals or search directions.

   (a) Are the vectors $\{d_k\}$ produced by this recursion Q-conjugate? Justify briefly (you may assume the standard Gram–Schmidt proof structure).

   (b) What happens when $Q = I$? Comment on what $\{d_k\}$ become in this special case.

   (c) **Programming.** Query the oracle `f2(srno, True)` to obtain A (this plays the role of Q) and b. Implement the **Conjugate Directions** method (name it `CD_SOLVE`) starting from $x_0 = 0$. Use the eigenvectors of A as the conjugate directions. At each iteration, compute and **print** the following quantities for the first 7 steps:

      i. the step size α_k,
      ii. the value $-\nabla f(x_k)^\top u_k$,
      iii. the corresponding eigenvalue λ_k.

      Submit: Your Python code and the printed values $(\alpha_k, -\nabla f(x_k)^\top u_k, \lambda_k)$ for the first 7 iterations.

2. (5 points) Query the oracle `f2(srno, True)` to obtain a symmetric positive definite matrix A (this plays the role of Q) and a vector b. Implement the Conjugate Gradient method (name the function `CG_SOLVE`), starting from $x_0 = 0$ to solve $Ax = b$.

   - Your `CG_SOLVE` function must support an additional optional argument `log_directions` (default = `False`).
   - When `log_directions=True`, the function should also return the first m residuals $r_0, \ldots, r_{m-1}$ and the CG search directions $p_0^{\mathrm{CG}}, \ldots, p_{m-1}^{\mathrm{CG}}$.
   - Using the residuals (or the CG raw direction vectors) as the sequence $\{p_k\}$, implement the Q-Gram–Schmidt recursion above (as a function `GS_ORTHOGONALISE`) with $Q = A$ to produce $\{d_k\}$.
   - **Submit:** Your Python code (`CG_SOLVE` with the `log_directions` option and `GS_ORTHOGONALISE`), the CG search directions $\{p_k^{\mathrm{CG}}\}$, the Gram–Schmidt Q-orthogonalised vectors $\{d_k\}$, and the number m of directions computed.

     You must submit *two separate .txt files*:

     (a) `plist_vwxyz.txt` – containing all CG search directions p_k^{CG},

     (b) `dlist_vwxyz.txt` – containing all Gram–Schmidt Q-orthogonalised vectors d_k.

     Each vector must be written one per row (use `np.savetxt` in Python). Do not truncate — print all components of every vector. This is required for grading. Reports without these files (or in the wrong format) will not be evaluated.

3. (2 points) Normalize each d_k in the A-inner product,

$$\tilde{d}_k \;=\; \frac{d_k}{\sqrt{d_k^\top A d_k}},$$

   and form the matrix M with entries

$$M_{ij} \;=\; \tilde{d}_i^\top A \tilde{d}_j.$$

   - **Submit:** the numeric matrix M (printed in your report).

4. (2 points) Compare the vectors $\{d_k\}$ produced by Q-Gram–Schmidt with the CG search directions $\{p_k^{\mathrm{CG}}\}$. Compute the A-inner-product cosine similarity for each corresponding pair:

$$\cos(\theta_k) \;=\; \frac{(p_k^{\mathrm{CG}})^\top A d_k}{\sqrt{(p_k^{\mathrm{CG}})^\top A p_k^{\mathrm{CG}}}\,\sqrt{d_k^\top A d_k}}.$$

   - **Submit:** Your code, and the list of cosine similarities $\cos(\theta_k)$ for $k = 0, \ldots, m-1$.

5. (1 point) Can you please conclude the purpose of this question in 1 line?

Oracle: Import the oracle `f2`. The arguments to be passed (strictly in the given order) are:

1. `srno`: the last five digits of your serial number passed as an integer, and,
2. `True`: which specifies that the oracle will return the data for this question.

The oracle will return:

- A symmetric positive definite (SPD) matrix A,
- A right-hand side vector b.

Function Specifications (for coding consistency)

- `CD_SOLVE`
  - **Function Name:** `CD_SOLVE`
  - **Inputs:**
    1. `A`: SPD matrix from oracle (NumPy array).
    2. `b`: Right-hand side vector (NumPy array).
    3. `x0`: Initial point (default = zero vector).
    4. `maxiter`: Maximum number of iterations (default = 100).
  - **Outputs:**
    1. `x`: Final iterate after Conjugate Descent.
    2. `alphas`: List of step sizes α_k.
    3. `numerators`: List of values $-\nabla f(x_k)^\top u_k$.
    4. `lambdas`: Corresponding eigenvalues λ_k.

- `CG_SOLVE`
  - **Function Name:** `CG_SOLVE`
  - **Inputs:**
    1. `A`: SPD matrix from oracle (NumPy array or `LinearOperator`).
    2. `b`: Right-hand side vector (NumPy array).
    3. `tol`: Convergence tolerance (default = 10^{-6}).
    4. `maxiter`: Maximum number of iterations (default = 10000).
    5. `log_directions`: Boolean flag (default = `False`). When set to `True`, the function must additionally return the first m residuals and search directions.
    6. `use_relative_tol`: Boolean flag (default = `False`). When set to `True`, the stopping rule becomes $\|r_k\|/\|r_0\| <$ `tol` (used in Question 2).
  - **Outputs (if `log_directions=False`):**
    1. `x`: Approximate solution vector.
    2. `iters`: Number of iterations taken.
    3. `residuals`: Residual norms $\|r_k\|_2$ at each iteration.

- **Outputs (if `log_directions=True`):**
  1. `x`, `iters`, `residuals` (same as above),
  2. `residual_list`: First m residuals $\{r_0, \ldots, r_{m-1}\}$,
  3. `directions`: First m CG search directions $\{p_0^{\mathrm{CG}}, \ldots, p_{m-1}^{\mathrm{CG}}\}$.

- `GS_ORTHOGONALISE`
  - **Function Name:** `GS_ORTHOGONALISE`
  - **Inputs:**
    1. `P`: A list (or array) of vectors $\{p_0, \ldots, p_{m-1}\}$ to be orthogonalised.
    2. `Q`: SPD matrix (here use A from oracle).
  - **Outputs:**
    1. `D`: The Q-orthogonalised vectors $\{d_0, \ldots, d_{m-1}\}$.

Both functions should be in your submission file and callable as:

```
# Conjugate Direction
x_cd, alphas, nums, lambdas = CD_SOLVE(A, b)

# Conjugate Gradient with logging
x, iters, residuals, r_list, p_list = CG_SOLVE(A, b, log_directions=True)

# Gram{Schmidt orthogonalisation
D = GS_ORTHOGONALISE(p_list, A)
```

Remark:

1. The same function `CG_SOLVE` will be reused in Question 2. For that question, you will simply call it with the default option `log_directions=False`, so make sure you write your implementation cleanly here.

2. Your implementation must treat A as a black-box operator. Do not assume that A is indexable; always use matrix–vector products such as `A @ x` or `A.dot(x)`. This ensures your code works both when A is a NumPy array (as in Question 1) and when A is a `scipy.sparse.linalg.LinearOperator` (as in Question 2).

QUESTION 2 (15 points)

1. (6 points) Use the oracle `f5(srno)` to obtain a symmetric positive definite (SPD) matrix A and a vector b. Implement the Conjugate Gradient method (starting from $x_0 = 0$) to solve $Ax = b$. Plot the residual norm $\|r_k\|_2$ against the iteration number k and report the number of iterations required to reduce the relative residual below 10^{-6}.

- **Submit:** Your CG implementation (function `CG_SOLVE`), the residual plot, and the iteration count.

2. (9 points) Now attempt to **improve the speed of convergence**. You may modify the algorithm in any way you think is appropriate. Compare the performance of your modified method against the standard CG implementation.

   - **Submit:** Your modified code (function `CG_SOLVE_FAST`), comparison plots of residual norms for both methods, iteration counts, and a short discussion (5–8 sentences) on what you tried and whether it improved convergence.

Oracle: Import the oracle `f5`. The argument to be passed is:

1. `srno`: the last five digits of your serial number passed as an integer.

The oracle will return:

- A: a 10000×10000 SPD matrix, provided as a `scipy.sparse.linalg.LinearOperator` (not a NumPy array). You cannot access its entries directly. Only matrix–vector products such as `A @ x` or `A.dot(x)` are supported.
- b: a right-hand side vector (NumPy array).

Function Specifications (for coding consistency)

- **Standard CG Implementation (reuse from Q1):**
  - **Function Name:** `CG_SOLVE`
  - *Note:* You must reuse the same `CG_SOLVE` function written for Question 1. For this question, call it with the default option `log_directions=False` and set `use_relative_tol=True` so that the stopping rule uses the relative residual criterion $\|r_k\|/\|r_0\| < 10^{-6}$.
- **Improved Implementation:**
  - **Function Name:** `CG_SOLVE_FAST`
  - **Inputs:** Same as `CG_SOLVE`.
  - **Outputs:** Same as `CG_SOLVE`.

Both functions should be in your submission file and callable as:

```
# Question 1 (standard CG with direction logging)
x1, iters1, res1, r_list, p_list = CG_SOLVE(A, b, log_directions=True)
# Question 2 (standard vs improved CG with relative residual criterion)
x2, iters2, res2 = CG_SOLVE(A, b, use_relative_tol=True)
x3, iters3, res3 = CG_SOLVE_FAST(A, b, use_relative_tol=True)
```

Remark: Your implementation must treat A as a black-box operator. Do not assume that A is indexable; always use matrix–vector products such as `A @ x` or `A.dot(x)`. This ensures your code works both when A is a NumPy array (as in Question 1) and when A is a `scipy.sparse.linalg.LinearOperator` (as in Question 2).

QUESTION 3 (7 points)

Consider the **Rosenbrock function**:

$$f(x_1, x_2) = (1 - x_1)^2 + 100(x_2 - x_1^2)^2,$$

which has a unique minimiser at $x^* = (1, 1)$.

1. (4 points) Implement Newton's method to minimise $f(x_1, x_2)$ using its analytic gradient and Hessian. Run the algorithm from the following four starting points:

$$(2, 2), \quad (5, 5), \quad (-10, -4), \quad (50, 60).$$

   - Plot the error $\|x_k - x^*\|_2$ versus iteration number k for each starting point (all four curves in one plot).
   - Produce **four separate contour plots** of the Rosenbrock function (in the region $[-10, 10]^2$), one for each starting point, with the Newton iterates overlaid (use markers to indicate the step sequence).
   - **Submit:** Your Python code, the error plot, and the four contour plots with Newton trajectories.

2. (3 points) Compare the behaviours observed.

   - Which starting points lead to rapid convergence to x^*?
   - Which ones fail or diverge?
   - Briefly explain why the starting point plays such a crucial role in Newton's method.
   - **Submit:** A short written explanation (5–6 sentences) interpreting your results.

Function Specifications (for coding consistency)

- **Newton Implementation:**
  - **Function Name:** `NEWTON_SOLVE`
  - **Inputs:**
    1. `f_grad`: Gradient function of $f(x)$ (callable).
    2. `f_hess`: Hessian function of $f(x)$ (callable).
    3. `x0`: Initial point (NumPy array of length 2).
    4. `tol`: Convergence tolerance (default $= 10^{-8}$).
    5. `maxiter`: Maximum number of iterations (default $= 100$).
  - **Outputs:**
    1. `x`: Final iterate.
    2. `iters`: Number of iterations performed.
    3. `trajectory`: List of iterates (for plotting Newton paths).

Your submission should include:

- The function `NEWTON_SOLVE`,
- A script that calls it from the four specified starting points,
- The error plot, and the four contour plots with Newton trajectories.

****