MMAE 350 Midterm 1 — Sample Cheat Sheet

(Two-column layout; print double-sided if allowed)

Python & Jupyter Essentials

Virtual environment: isolates packages per course/project; improves reproducibility.
Jupyter notebook: best for mixing derivations, code, plots, and short write-ups.
Common workflow: activate env → start Jupyter → run cells top-to-bottom.
Indexing reminder:

`range(n)` : $0, 1, \ldots, n-1$ `range(1,n+1)` : $1, 2, \ldots, n$

Loops:

- `for` loop: repeat a known number of steps
- `while` loop: repeat until a condition is met

Conditionals:

- Two cases: `if / else`
- Three cases: `if / elif / else`

NumPy idea: use arrays + vectorized operations for speed (vs. Python loops).

Tiny Pseudocode Patterns

Sum of a vector x:

```
s = 0
for i = 1..n:
    s = s + x[i]
```

Count positives in vector x:

```
count = 0
for i = 1..n:
    if x[i] > 0:
        count = count + 1
```

Extract diagonal of matrix A:

```
for i = 1..n:
    d[i] = A[i,i]
```

Matrix Algebra Basics

Shapes: $A \in \mathbb{R}^{m\times n}$, $x \in \mathbb{R}^n$, $Ax \in \mathbb{R}^m$.
Matrix–vector product:

$$b_i = \sum_{j=1}^{n} A_{ij} x_j$$

Transpose: $(A^T)_{ij} = A_{ji}$.
Invertible: unique solution to $Ax = b$ for every b; $\det(A) \neq 0$.
Avoid explicit inverse: compute x by solving $Ax = b$ (more stable, faster) rather than $x = A^{-1}b$.

Matrix–Vector Multiply (Nested Loops)

Pseudocode:

```
for i = 1..n:
    b[i] = 0
    for j = 1..n:
        b[i] = b[i] + A[i,j]*x[j]
```

NumPy: `b = A @ x`

Solving Linear Systems: Big Picture

Solve $Ax = b$.
Direct methods: finite number of steps (Gaussian elimination; Thomas for tridiagonal).
Iterative methods: repeated sweeps (Gauss–Seidel).
Residual:

$$r = b - Ax$$

Stop when $\|r\|$ is small (or when max change in x is small).

Gaussian Elimination (Outline)

Goal: transform to upper-triangular U then back substitute.
Forward elimination idea:
Use row operations to zero out entries below the pivot A_{kk}.
Back substitution:

$$x_n = \frac{b_n}{U_{nn}}, \qquad x_i = \frac{b_i - \sum_{j=i+1}^{n} U_{ij} x_j}{U_{ii}}$$

Thomas Algorithm (Tridiagonal)

For tridiagonal A with subdiag a_i, diag d_i, superdiag c_i.
Forward sweep: define modified coefficients d_i', c_i', b_i'.

$$d_1' = d_1, \qquad c_1' = \frac{c_1}{d_1'}, \qquad b_1' = \frac{b_1}{d_1'}.$$

For $i = 2, \ldots, n$:

$$d_i' = d_i - a_i\, c_{i-1}', \qquad c_i' = \frac{c_i}{d_i'}, \qquad b_i' = \frac{b_i - a_i\, b_{i-1}'}{d_i'}.$$

Back substitution:

$$x_n = b_n', \qquad x_i = b_i' - c_i' x_{i+1} \quad (i = n-1, \ldots, 1).$$

Key fact: $O(n)$ work (very fast).

Gauss–Seidel (Idea + One Sweep)

Idea: use newest values immediately as you sweep through unknowns.
Update form:

$$x_i \leftarrow \frac{1}{A_{ii}}\left(b_i - \sum_{j\neq i} A_{ij}x_j\right)$$

One-sweep pseudocode:

```
for i = 1..n:
    sigma = 0
    for j = 1..n, j != i:
        sigma += A[i,j]*x[j]
    x[i] = (b[i]-sigma)/A[i,i]
```

Convergence helpers: diagonal dominance often helps:

$$|A_{ii}| \geq \sum_{j\neq i} |A_{ij}|$$

Nonlinear Equations: Newton's Method (1D)

Solve $f(x) = 0$.
Linearization (Taylor):

$$f(x) \approx f(x_k) + f'(x_k)(x - x_k)$$

Newton update:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

Residual: $r_k = |f(x_k)|$.
Quadratic convergence (conditions): The function has a well-defined derivative near the root, the derivative at the root is not zero, and the initial guess is sufficiently close to the true solution.

Newton Pseudocode (Scalar)

```
given f, fprime, x0, tol, max_iter
x = x0
for k = 1..max_iter:
    fx = f(x)
    if abs(fx) < tol:
        stop
    fpx = fprime(x)
    x = x - fx/fpx
end
```

Common failure modes: bad initial guess; derivative near zero; divergence.

Newton for Systems (Memory Form)

Solve $F(x) = 0$ with $x \in \mathbb{R}^n$.
Linear solve each iteration:

$$J(x_k)\,\Delta x = -F(x_k)$$

$$x_{k+1} = x_k + \Delta x$$

Pseudocode:

```
x = x0
for k = 1..max_iter:
    r = norm(F(x))
    if r < tol: stop
    J = Jacobian(x)
    solve J*dx = -F(x)
    x = x + dx
end
```

SymPy → NumPy Reminder

SymPy: define symbolic f, compute f', simplify.
NumPy: evaluate numerically inside loops for speed.
Typical workflow:

- Symbolic: build $f(x)$ and $f'(x)$ in SymPy
- Convert: create numerical callables (e.g., `lambdify`)
- Iterate: run Newton (or GS) numerically