C Mathematical Programming – Exam Cheat Sheet

Focused on C: floating-point, core library functions, data structures, and BLAS/LAPACK. Each block lists the required #include and a tiny usage example.

1. Floating-Point & Numerical Stability

Headers: `#include <math.h> #include <float.h> #include <math.h>`

1. log1p(x): Accurate log(1 + x) for small x.
2. expm1(x): Accurate exp(x) - 1 for small x.
3. hypot(x, y): Stable sqrt(x*x + y*y).
4. fma(x, y, z): Compute x*y + z with a single rounding.
5. frexp(x, &exp;), ldexp(x, exp), scalbn(x, n): Base-2 decomposition and scaling.
6. nextafter(x, y): Next representable double after x toward y.
7. DBL_EPSILON, DBL_MAX, DBL_MIN: Machine epsilon and limits (float.h).
8. isfinite(x), isnan(x), fpclassify(x): Classify floating-point values.

Example: stable norm and machine epsilon

```
#include <math.h>
#include <float.h>

double x = 1.0, y = 1e-300;
double r = hypot(x, y);      /* stable sqrt(x*x + y*y) */
if (fabs(x + DBL_EPSILON - x) > 0.0) {
    /* DBL_EPSILON is the distance to next representable double */
}
```

2. Core C Utilities for Numerical Software

Headers: `#include <stdlib.h> #include <string.h> #include <assert.h>`

1. malloc, calloc, realloc, free: Dynamic arrays and matrices.
2. memcpy, memmove, memset: Efficient block memory operations.
3. qsort, bsearch: Standard sorting and binary search.
4. assert: Runtime checks of assumptions.

Example: dynamic vector and qsort

```
#include <stdlib.h>
#include <string.h>
#include <assert.h>

int cmp_double(const void *a, const void *b) {
    double da = *(const double *)a;
    double db = *(const double *)b;
    return (da > db) - (da < db);   /* -1, 0, or 1 */
}

size_t n = 100;
double *v = malloc(n * sizeof *v);
assert(v != NULL);
/* ... fill v ... */
qsort(v, n, sizeof *v, cmp_double);
free(v);
```

3. Data Structures in Numerical Computing

Typical C representations:

1 Arrays: dense vectors/matrices, e.g. double a[m*n]; with index a[i*n + j].

2 Linked lists: structs with next pointers; flexible but slow for numeric kernels.

3 Sparse matrices: CSR/CSC using index and value arrays.

Example: row-major dense matrix access

```
int m = 3, n = 4;
double *A = malloc(m * n * sizeof *A);

/* A[i,j] at row i, column j: */
int i = 1, j = 2;
A[i*n + j] = 1.0;
```

4. Algorithmic Considerations

1 Time: know typical costs: O(n), O(n log n), O(n*n).

2 Space: prefer contiguous arrays for cache locality.

3 Iterative vs recursive: in C numerical code, prefer iterative.

Example: iterative vs recursive sum

```
double sum_iter(const double *x, size_t n) {
    double s = 0.0;
    for (size_t i = 0; i < n; ++i) s += x[i];
    return s;
}
```

5. BLAS – Basic Linear Algebra Subprograms (double precision)

Header (C interface, implementation-dependent): `#include <cblas.h>`

Level 1 – vector:

1 cblas_daxpy(n, alpha, x, incx, y, incy): y = alpha*x + y.

2 cblas_ddot(n, x, incx, y, incy): dot product.

3 cblas_dnrm2(n, x, incx): Euclidean norm.

4 cblas_dscal, cblas_dcopy: scale and copy vectors.

Level 2 – matrix-vector:

1 cblas_dgemv(…): y = alpha*A*x + beta*y.

2 cblas_dger(…): rank-1 update A = alpha*x*y^T + A.

Level 3 – matrix-matrix:

1 cblas_dgemm(…): C = alpha*A*B + beta*C.

2 cblas_dtrsm(…): solve triangular systems AX = B or XA = B.

Example: matrix-vector multiply with cblas_dgemv (column-major)

```
#include <cblas.h>

int m = 3, n = 2;
double A[6] = { /* column-major: A(0,0),A(1,0),A(2,0),A(0,1),... */ };
double x[2] = {1.0, 2.0};
```

```
double y[3] = {0.0, 0.0, 0.0};

cblas_dgemv(CblasColMajor, CblasNoTrans,
            m, n, 1.0,
            A, m,
            x, 1,
            0.0,
            y, 1);
```

6. LAPACK – Linear Algebra Solvers

Header (C interface, implementation-dependent): `#include <lapacke.h>`

1. LAPACKE_dgesv: Solve A*x = b using LU factorization.
2. LAPACKE_dgetrf / LAPACKE_dgetrs: LU factor and solve.
3. LAPACKE_dpotrf / LAPACKE_dpotrs: Cholesky for SPD matrices.
4. LAPACKE_dgels: Least-squares problems.
5. LAPACKE_dsyev: Eigenvalues/eigenvectors of symmetric matrices.

Example: solve A x = b with LAPACKE_dgesv (column-major)

```
#include <lapacke.h>

int n = 3, nrhs = 1, lda = 3, ldb = 3, info;
int ipiv[3];
double A[9] = { /* column-major A */ };
double b[3] = { /* right-hand side */ };

info = LAPACKE_dgesv(LAPACK_COL_MAJOR, n, nrhs,
                     A, lda, ipiv,
                     b, ldb);
/* On success, b contains the solution x */
```

7. Testing, Debugging & Robustness

1. Check residuals: norm(A*x - b) or max absolute error.
2. Use small problems with known analytic solutions.
3. Check for isnan / isinf in intermediate results.
4. Use assert and defensive programming on indices and sizes.

Key idea: in C numerical code, rely on standard headers and tested BLAS/LAPACK implementations instead of rewriting basic operations.