|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Computer arithmetic & algorithms | | dmdm-1...d0.d-1d-2...d-n = dm x 10m + dm-1 x 10m-1 + ...+ d0 x 100 + d-1 x 10-1 + d-2 x 10-2 + ...+ d-n x 10-2  bmbm-1...b0.b-1...b-n = bm x 2m + bm-1 x 2m-1 + ...+ b0 x 20 + b-1 x 2-1+ ... + b-n x 2-n | | | | base 10 (decimal),  dmdm-1...d0: integer, d-1d-2...d-n: fractional  converting binary to decimal |
| dmdm-1...d0.d-1d-2...d-n = (brbr-1...b0.b-1b-2...)2  dmdm-1...d0 = (brbr-1...b0)2 & d-1d-2...d-n = (b-1b-2...)2  To find integer part, - divide dmdm-1...d0 by 2 (remainder = b0)  - divide quotient by 2 (remainder = b1)...repeat process until quotient is 0  For fractional part, - multiply .d-1d-2...d-n by 2 (integer part = b-1)  - multiply new fractional part by 2 (integer part = b-2)...repeat until fractional part is 0; otherwise fractional part is infinite | | | | converting decimal to binary  integer = integer & fractional = fractional  E.g. 2.4 = 10.011001100110...  2/2 = 1R0. 1/2 = 0R1  0.4 x 2 = 0 + 0.8 0.8 x 2 = 1 + 0.6  0.6 x 2 = 1 + 0.2 0.2 x 2 = 0 + 0.4  0.01100110... = 2-2 + 2-3 + 2-6 + 2-7 + ... =  (2-2 + 2-3) = = 0.4 |
| Floating-point formats | | Binary num in scientific notation normalized form | (brbr-1...b0.b-1...b-n)2 = ±(1.br-1...b0b-1...b-n)2 x 2r (0.001012 = 1.012 x 2-3) | | | |
| ±(1.s1s2...sN)2 x 2k  Double precision floating point: ±(1.b1b2...b52)2 x 2k | | |  |  |  |  | | --- | --- | --- | --- | | IEEE standard | Number of bits | | | | precision | sign | exponent | mantissa (N) | | double | 1 | 11 | 52 | | | |
| machine epsilon = dist btw 1 and smallest floating point num > 1 and for double precision, = 2-52 | | | | |
| Rounding rule | | IEEE Rounding to Nearest Rule  1. 53rd bit = 0: truncate after 52nd bit  2. 53rd bit = 1:  a. 54th bit onwards all 0 & 52nd bit = 0: truncate after 52nd bit  b. else 1 added to 52nd bit | | | | Applies to both normal and subnormal num  (1.\*...\*52053)2 x 2k = (1.\*...\*52)2 x 2k  (1.\*...\*052153)2 x 2k = (1.\*...\*052)2 x 2k  (1.\*...\*152153)2 x 2k = (\*.\*...\*052)2 x 2k  (1.\*...\*\*52153...\*1\*...)2 x 2k = (\*.\*...\*\*52)2 x 2k |
| Subnormal floating point | | For num smaller than 2-1022 ≈ 2.2 x 10-308, subnormal floating point number is used: ±(0.s1s2...s52)2x 2-1022 | | | | So, smallest +ve double precision num = (0.00...1)2 x 2-1022 = (1 x 2-52) x 2-1022 = 2-1074 |
| Although num below is machine representable, adding to 1 may have no effect | | | | |
| Computer arithmetic | | For a num x, fl(x) = num stored in computer (not exact) | | | Abs error = |p\*-p|. Relative error = ≤ | |
| Use special symbol to represent computer arithmetic | | | | xy = fl(fl(x) + fl(y)) xy = fl(fl(x) - fl(y))... |
| Matrix multi-plication | | A = (aij)m x n , B = (bij)n x p , C = AB = (cij)m x p  cij = , i = 1,...,m, j = 1,...,p  O(mpn). Exact is O(mpn + mp(n-1)) = O(mp(2n-1)) | | | 3 main qns: Find i, k s.t. aik ≠ 0. Find k, j s.t. bkj ≠ 0. Find i, j s.t. cij ≠ 0 | |
| Lower triangular multi-plication | Suppose A and B are lower triangular n x n matrix, C = AB also lower triangular  cij = 0 i < j, C = (cij)n x n  For i ≥ j: – A is lower triangular: aik = 0, i < k  – B is lower triangular: bkj = 0, k < j  – aikbkj = 0 if i < k or k < j. So multiplication only needed for k ≤ i or k ≥ j  cij = , 1 ≤ j ≤ i ≤ n | | | | | = (n - 1) + 1 |
| = = num of terms \* (1st term + last term) / 2 |
| = |
| Upper Hessen-berg multi-plication | A = (aij)n x n be tridiagonal matrix = , C = (cij)n x n  B be upper triangular and C = AB be upper Hessenberg matrix  cij = 0, j < i - 1, 3 ≤ i ≤ n  For j ≥ i - 1: – aik = 0, k < i - 1, 3 ≤ i ≤ n or k > i + 1, 1 ≤ i ≤ n-2  – bkj = 0, k > j  – aikbkj = 0, k < i - 1, 3 ≤ i ≤ n or k > min(i+1, j), 1 ≤ i ≤ n - 2  ≠ 0, i - 1 ≤ k ≤ min(i+1,j),i = 1,...,n, j = max(1,i-1),..., n  cij = , i = 1,...,n, j = max(1,i-1),..., n | | | | | |

|  |  |  |
| --- | --- | --- |
| Bisection Mtd | Intermediate Value Theorem (IVT): Let f be cts fn on [a,b], where f(a)f(b) < 0. Then r (a,b) s.t. f(r) = 0  Bisection mtd: Solve for eqn: f(r) = 0. Keep dividing interval by 2 until length of new interval/2 ≤ TOL. Soln: exact root r = approx root ± TOL  After n bisection steps: Approximate root = midpoint of (an, bn) = (an + bn)/2  Error of approx soln = |exact root - approx root| ≤ = =  Num of fn evaluations = n + 2 (f(a), f(b), n times of f(c)). Convergence rate = 1/2; need predefine initial interval. Convergence guaranteed | |
| Soln is correct within p d.p if error is less than 0.5 x 10-p. Num of steps for bisection: < 0.5 x 10-p | |
| Fixed-Point Iteration (FPI) | Solve eqn: g(x) = x, by iterating xi+1 = g(xi). r is a fixed pt if g(r) = r  Suppose f(x) = g(x) - x, and xa is approximation for r where f(r) = 0 | Backward error = |g(xi+1) - xi+1| = |f(xa)|. Forward error = |r-xa|  Method relies on Fixed-Point Thrm |
| Convergence Thrm: If |g'(r)| < 1: will converge with rate S = |g'(r)|  So S = 0: fastest convergence rate. S < 1: cfm converge. S > 1: won't converge. S = 1: may or may not | |
| Horner's Mtd | Horner's Mtd: most optimal mtd for finding value of polynomial. O(m). Given x, evaluate h(x). Common that h is polynomial.  Pm(x) = ao + a1x + ... + am-1xm-1 + amxm = a0 + x(a1 + x(a2 + ... + x(am-1 + xam)))  pm = am. pm-1 = am-1 + xpm. ... p1 = a1 + xp2. p0 = a0 + xp1 | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Gaussian Elimina-tion  O(n3) Leading term: | | An x nxn x 1 = bn x 1 . Note lower triangular part not set to zero as they would not be used in further computation  Subtraction: (without labelling rows). Multiplication: (without labelling rows)  Division: n(n+1)/2 (without labelling rows). Note zero column = non-invertible  Could have cases where = 0. So label rows to keep track of swapping rows  Due to computer arithmetic errors, num supposed to be 0 becomes a very small num and cause loss of significant digits  Backward sub: O(n2) | | | |
| Partial Pivoting | | 1st mtd to perform row swap to minimise errors  Select pivot elem s.t. its absolute value is largest in a particular col. Num of comparisons is = =  However, if remaining entries of pivot row also has large magnitude, then would cause loss of significant digits in the other rows. | | | |
| Relative absolute ratio | | A way to do this is to compare the relative absolute sizes in each col j = 1,...,n ,  relative absolute value of row i = . Divisions = , Comparisons =  By choosing largest relative absolute size for each col as pivot elem, loss of sig digits is minimised | | | |
| Scaled Partial Pivoting (SPP) | | 2nd mtd to perform row swap to minimise errors. ≤ : don't swap. > : swap  To save computational cost, assume max entry of row does not change too much in elimination process, and only find max for each row once at beginning, then use this max from original row to calculate relative absolute ratio: scaled partial pivoting  Comparisons = . Note max(row i) stay fixed even if row swap is performed | | | |
| LU factoriza-tion  O(n3) + O(2pn2) | | Solve for multiple L.S with same coefficient matrix, Ax1 = b1, Ax2 = b2..., Axp = bp. So can preprocess A to not repeat ops.  Find A = LU by Gaussian elimination w/o pivoting strategies, where L is lower triangular, U: upper triangular. LUx = b.  L = strictly lower part of processed A (multipliers mji) + diagonal all 1. U = upper triangular of processed A | | | |
| Forward substitution (solve Ly = b for y). Backward substitution (solve Ux = y for x)  Time complexity for both substitution = O(n2). So, solving p LS = O(2pn3) | | | |
| PA = LU factoriza-tion  O(n3) + O(2pn2) | | Matrix which require row swap to get REF cannot be LU factorized. (or can check det of top left entry, top left 2 x 2 entries, top left 3 x 3 ... ≠ 0 then can be factorized). Hence, need perform row swap at start with SPP, PAx = Pb, where P is a permutation matrix (n x n matrix consisting all 0, except for a single 1 in every row and col)  Now PA = LU. L = strictly lower part of processed A (multipliers mji) + diagonal all 1. U = upper triangular of processed A  Solve Ly = Pb, then solve Ux = y. No need to explicitly find P, just output r (stored row index) to replace P | | | |
| A = LU & PA = LU | | | | If n is large, memory to store L and U might be consuming | |
| Special matrix | Symmetric positive-definite matrix. n x n matrix A is symmetric if AT = A. A is positive-definite if xTAx > 0 col vector x ≠ 0.  To check if matrix is positive-definite, could expand algebraically with x and then complete the sq to check > 0.  OR If A is symmetric, A is positive-definite iff all eigenvalues > 0  Principal submatrix of sq matrix A is a sq submatrix whose diag entries = diag entries of A. (a11, a22, a33, , , )  Any principal submatrix of a symmetric positive-definite matrix is symmetric positive-definite. | | | | |
| Cholesky factoriza-tion  O(n3) + O(pn2) | | Cholesky factorization: every symmetric positive-definite matrix A can be factored as A = RTR (R is upper triangular). Hence would save roughly half the memory compared to A = LU. Idea: Use row/col ops to reduce A into identity matrix to get A = RTR.  Forward sub: RTy = b for y. Backward sub: Rx = y for x. O(n2) | | | |
| A = k = 1: R = , A = ,  k = 2: R = , A =  Repeat until k = n, then A becomes I so A = RTR. Note also positive definite. | | | |
| Strictly Diagonally Dominant | | | A is stricly diagonally dominant (sdd) if for each 1 ≤ i ≤ n, |aii| > , i.e. diagonal entry > sum of non-diagonal entries in same row, then Jacobi and Gauss-Seidel mtd will converge  If A sdd, then A is a nonsingular matrix. If A not sdd, MIGHT still converge (check spectral radius) | | |
| Spectral Radius | | | Another mtd to check convergence is spectral radius p(B) = max magnitude of eigenvalues of B. If p(B) < 1, and c is arbitrary, then for any vector x0, x­k+1 = Bxk + c converges. In particular, check p(D-1(L + U)) < 1. Use det(D-1(L + U) - I) = 0 to find eigenvalues.  Note determinant of matrix = product of eigenvalues | | |
| Jacobi Method  O(n2) | | | = + + = L + D + U | | |
| Similar to fixed-point iteration. An x n = L + D + U. Ax = b. (L + D + U)x = b. Dx = b - (L + U)x. x = D-1(b- (L + U)x).  D-1 is just reciprocal of all entries in D. So x(k+1) = D-1(b - Lx(k) - Ux(k))  Then solve eqn element-wise, i = 1,...,n: =  So calculation can be parallelized | | |
| Gauss-Seidel Mtd O(n2) | | | Similar to Jacobi. Now, x(k+1) = D-1(b - Lx(k+1) - Ux(k))  i = 1,...,n: =  Now cannot parallelized, as x2,... xn dependent on x1, ..., xn-1. But since updated values are used, will converge faster than Jacobi | | |
| Successive Over-Relaxation (SOR) Mtd O(n2) | | | | Let be a real num, and x(k+1) = (1 - )x(k) + D-1(b - Lx(k+1) - Ux(k)). is called relaxation parameter and >1 = over-relaxation  i = 1,...,n: = (1 - ) +  GS: = 1. SOR: > 1. Need to choose wisely, usually 1.1 or 1.2. Faster convergence than GS | |
| When to use which mtd? | | Direct mtd: Gaussian elimination, A = LU or PA = LU, Cholesky factorization. O(n3) for preprocessing, O(n2) for finding  Iterative mtd: Jacobi, Gauss-Seidel, SOR. O(n2) | | | Use iterative mtd if 1. requirement of accuracy not high, save computational cost  2. good approximation already known (to be used as initial guess)  3. If A is sparse (many entries = 0). Most expensive op is matrix-vector multiplication which would be cheaper |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Interpola-tion | | Given data points (x0, f(x0)), (x1, f(x1)), ..., (xn, f(xn)). Want to find polynomial of deg n, Pn(x) to connect these pts to restore original fn f, i.e. Pn(xi) = f(xi), i = 0,1,...,n | | Weierstrass approximation theorem: Let f be a cts fn on [a, b]. For any > 0, a polynomial P(x) s.t. |f(x) - P(x)| < , x [a, b] |
| 1. Gaussian Elmt | | Write Pn(x) = a0 + a1x + ... + anxn  From general eqn for polynomial, sub data points in to get n+1 eqns and solve L.S to find coeff | | |
| Basis fn | | Basis fn: {(x), (x), ..., (x)}, where (xj) = , | | Kronecker delta, = , j = 0,1,..., n, k = 0,1, ..., n |
| 2. Lagrange polynomial | | Lk(x) = = . So Lk(xj) = . Lk(x) = kth Lagrange basis polynomial  Then Pn(x) = f(x0)L0(x) + f(x1)L1(x) + ... + f(xn)Ln(x) . Pn(x) = Lagrange interpolating polynomial of deg n | | |
|  | LS is guaranteed to have a soln if (deg n, num of eqns m)  m=n: By constructing Lagrange interpolating polynomial  m > n: Not guaranteed  m < n: Infinitely many solution | | | Using LS: Pros – direct mtd | Cons: tedious computations  Lagrange: Pros ­– easy to analyse. Good if we need to interpolate many fns with same set of interpolating nodes | Cons: not convenient to add more data points |
| Uniqueness of Lagrange polynomial | | | If x0, x1, ..., xn are n + 1 distinct nums and f is a fn whose values are given at these nums,  then a unique polynomial Pn(x) of degree at most n exists with f(xk) = Pn(xk), for k = 0,1,...,n. Pn(x) = Lagrange polynomial | |
| Adding more inter-polating nodes | Let Pn-1 be the Lagrange interpolating polynomial of f(x) with n nodes x0, x1,...,xn-1. Suppose we get one more data point (xn, f(xn)).  Let Qn(x) = Pn(x) - Pn-1(x). Qn(x) is the unique interpolating polynomial that interpolates (x0, 0), (x1, 0), ..., (xn-1, 0), (xn, f(xn - Pn-1(xn))  Qn(x) = f[x0, x1, ..., xn](x - x0)(x - x1)...(x - xn-1) where f[x0, x1, ..., xn] = nth divided diff of f =  So Pn(x) = Pn-1(x) + Qn(x) = Pn-1(x) + f[x0, x1, ..., xn](x - x0)(x - x1)...(x - xn-1) = ... = P0(x) + f[x0, x1](x - x0) + ... + f[x0, ..., xn](x - x0)(x - x1)...(x - xn-1) | | | |
| 3. Newton's Poly-nomial (Easier to add more nodes) | Pn(x) = = P0(x) + f[x0, x1](x - x0) + ... + f[x0, ..., xn](x - x0)(x - x1)...(x - xn-1)  And nth divided diff of f = f[x0, x1, ..., xn] = . where f[x0] = P0(x) = f(x0).  Note. order of nodes don't matter. i.e. f[x0, x1, x2] = f[x1, x2, x0]  Let x0, ..., xn be n+1 distinct real nums. Then f[x0, x1, ..., xn] = . Compute f[x0],…,f[xn] first, then f[x0, x1] = ...  O(n2) for computing nth divided diff. Num of entries = n + (n-1) + ... + 1 = . So total need n(n+1) subtraction and divisions  To compute Pn(x) for some x, use Horner's method. O(n)  P f[x0, x1, ..., xn]; for k = n-1,...,0 do {P f[x0, x1, ..., xk] + (x-xk)P} | | | |
| Error of interpola-tion | Runge's phenomenon: Wider oscillation at ends (i.e. worse interpolation at ends). num of nodes only worsen approximation at ends.  Error of interpolation, |f(x) - Pn(x)| : error is 0 on all nodes.  gn(x) = |(x-x0)(x-x1)...(x-xn)| = has similar pattern to error |f(x) - Pn(x)| w equally-spaced nodes  Let x0 < x1 <...< xn be n+1 distinct pts on [a, b]. If f Cn+1([a,b]), i.e. all derivatives f, f(1),..., f(n+1) are cts in [a, b], and Pn is the interpolating polynomial of f w deg ≤ n at x0, x1, ..., xn. Then x , dependant on x0, ..., xn, x, and min and max of {x0, x1, ..., xn, x}, s.t.  f(x) - Pn(x) = (x-x0)(x-x1)...(x-xn) | | | |
| Cheby-shev Interpola-tion | Chebyshev nodes: xk = cos, k = 0,1,..., n in [-1, 1]. Let Tn(x) := (deg n, n order) Chebyshev polynomial = cos((n)arccos x) (n nodes)  Using Chebyshev nodes means = Tn+1(x) ≤ is the smallest.  So now gn(x) = | cos((n+1)arccos x)| ≤ . gn 0 as n ∞. Error of interpolation for Chebyshev = f(x) - Pn(x) = Tn+1(x) ≤  For [a,b]: xk = + cos, k = 0,1,..., n. ≤  By using Chebyshev nodes to interpolate w Lagrange/divided diff, this polynomial = Chebyshev interpolating polynomial w deg n-1  Using Chebyshev nodes, error is worse ard center, but much milder near ends (error more evenly distributed). As n , Chebyshev f(x) | | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Linear Least Square Problem | Inconsistent sys: SLE w no solution, typically m ≥ n  Length = Euclidean norm = ||x||2 =  Dot product of 2 n-dimensional col vectors = uv = uTv = u1v1 + u2v2 + ... + unvn  If uTv = 0, then u and v are perpendicular/orthogonal to each other. uv | | | | | Note xTx = ||x||2. Normal equations: ATA = ATb  is L.S sol to Ax = b, which minimizes Euclidean norm of the residual r = b - Ax  If r = 0, then is the sol to Ax = b. |
| If need to use Am x n (full col rank = cols all LI) for multiple inconsistent sys, can use Cholesky factorization for ATA = RTR, where R is an upper triangular matrix | | | ATA is symmetric positive-definite. For all x ≠ 0, xTATAx = (Ax)T(Ax) = yTY = ||y||2 > 0  Forward sub: RTy = ATb. Backward sub: R = y | | |
| QR factorization | | ATA not numerically stable as could have rounding errors. A = QR: (m x n) = (m x n)\*(n x n)  So, use (reduced) QR factorization: Am x n = Qm x nRn x n, where Q is an orthogonal matrix (QT = Q-1), R is upper triangular. Then Rx = QTb.  Since Q is orthogonal matrix, it has orthonormal cols (i.e. ||**q**i|| = 1, **q**j = 0 if i ≠ j)  =  = | | | | |
| Gram-Schmidt orthogonalization | | | **y**1 = **a**1, r11 = ||**y**1||, **q**1 =  r12 = **a**2, **y**2 = **a**2 - r12**q**1, r22 = ||**y**2||, **q**2 = | | **y**j = **a**j - r1j**q**1 - r2j**q**2 - ... - rj-1,j**q**j-1, **q**j =  rij = **a**j, rjj = ||yj||, | |

­­­­

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Taylor's Thrm | Suppose f: is n+1 times differentiable on some open interval w the nth derivative f(n) cts on [a, x]. Then c [a,x] s.t.  f(x) = f(a) + f'(a)(x-a) + f''(a)(x-a)2 + ... + f(n)(a)(x-a)n + f(n+1)(c)(x-a)n+1 = P(x) + approximation error  The nth order Taylor polynomial for f at a is P(x) = f(a) + f'(a)(x-a) + f''(a)(x-a)2 + ... + f(n)(a)(x-a)n, w approximation error = f(n+1)(c)(x-a)n+1 | | | | |
| 2-point forward-diff formula | | | | | If f is twice continuously differentiable, then by Taylor's thrm, let x = x+h, a = x,  then f'(x) = – f''(c) where c [x,x+h] (w error term). 1st order method since error f''(c) is O(h) |
| Generalized IVT | | | | | Let f be a cts fn on interval [a,b]. Let x1, ..., xn [a,b] and a1, ..., an > 0. Then c [a,b] s.t. (a1 + ... + an)f(c) = a1f(x1) + ... + anf(xn) |
| 3-point centered-diff formula | | | | | Expand f(x+h) and f(x-h) to f''' w Taylor to get f'(x) = – h2f'''(c), where x-h < c < x+h  2nd order method since error h2f'''(c) is O(h2), approx is better. Generally, higher order appox formula more accurate |
| Approx formula for higher derivatives | | | | For f''(x), f(x+h) = f(x) + hf'(x) + f''(x) + f'''(x) + f(4)(c1) & f(x-h) = f(x) - hf'(x) + f''(x) - f'''(x) + f(4)(c2) where x-h < c2 < x < c1 < x+h  Adding both, 3-point centered diff formula for f''(x) ≈ w error f(4)(c) where x-h < c < x+h | |
| Rounding error | | Let (x+h), (x-h) be floating-point version of f(x+h), f(x-h), i.e. f(x+h) = (x+h) + , f(x-h) = (x-h) + , for some machine rounding error , . Then error in approx for 3-point centered-diff = |f'(x) - '(x)| ≤ , where , < > 0 and |f'''(c)| ≤ M. Smallest error at h = | | | |
| Extrapolation | | | order n formula for approximating Q: Q = Fn(h) + K(h)hn, where K(h) depends on h but can be treated as constant over range of h  (Richardson) extrapolation: Fn+1(h) + O(hn+1) = + O(hn+1), where Fn+1(h) is at least an order n+1 formula | | |

|  |  |  |  |
| --- | --- | --- | --- |
| MVT for integrals | | Let f be cts fn and g be an integrable fn that does not change sign on [a,b]. Then c [a,b] s.t. = f(c) | |
| Newton-Cotes (NC) approach | Trapezoid Rule: = (y0 + y1) - f''(c) := approx + error where h = x1 - x0 and c [x0, x1] | | |
| Simpson's Rule: = (y0 + 4y1 + y2) - f(4)(c) := approx + error where h = x1 - x0 = x2 - x1 and c [x0, x2] | | |
| Deg of precision | Deg of precision of a numerical integration mtd is the greatest int k for which all deg k or less polynomials are integrated exactly by the mtd.  Deg of precision of Trapezoid rule = 1. Deg of precision of Simpson Rule = 3. | | |
| Composite NC Formulas | Composite Trapezoid Rule: = (y0 + 2 + ym) - f''(c) where h = and c [a,b]. 2nd order mtd since O(h2).  Composite Simpson's Rule: = (y0 + 4 + 2 + y2m) - f(4)(c) where h = and c [a,b]. 4th order mtd | | |
| Open Newton-Cotes Mtd | Use if fn not valid on endpoints. Applicable for fn whose f'' is cts on [a,b]  Midpoint Rule: = hf(w) + f''(c) where h = (x1 - x0), w is midpoint := x0 + h/2 and c [x0,x1].  Composite Midpoint Rule: = h + f''(c) where h = and c [a,b]. 2nd order mtd since O(h2) | | |
| Romberg integration | So F2k(h) := | | |
| Text, whiteboard  Description automatically generated | | Text  Description automatically generated with low confidenceRomberg triangle:  A picture containing text  Description automatically generatedR11, R22, ... are just normal composite trap rule w 1,2,... panels |