|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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   |  |  |  |  | | --- | --- | --- | --- | | IEEE standard | Number of bits | | | | precision | sign | exponent | mantissa (N) | | single | 1 | 8 | 23 | | double | 1 | 11 | 52 | | long double | 1 | 15 | 64 | | | | Computer use bit (binary digit 0 or 1) to store info about the digits / mantissa (s1, s2,...sN), exponent k (in binary form as well) and sign (0 for +ve and 1 for -ve)  Note that exponent will use 1 bit for its sign and the other bits for its binary form  -Use double precision for this module | | | | |
|  | | Smallest +ve normalized double precision floating-point num: +(1.00...00 x 10-1111111110)2 = 1 x 2-1022 ≈ 2.22 x 10-308  +(1.00..00 x 10-1111111111) reserved for +∞  Largest +ve normalized double precision floating-point num: +(1.11...11 x 10+1111111111)2 = (2 - 2-52) x 21023 ≈ 1.8 x 10308  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) | | | | | | |  |
| 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)) | | | | | | | initialise C = (cij)m x p to be zero matrix  for i = 1,..., m do  for j = 1,...,p do  cij ai1b1j  for k = 2,..., n do  cij cij + aikbkj  Result: C = (cij)m x p. |
|  | | 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 | | | | | | | | initialise cij = 0 for all 1 ≤ i < j ≤ n  for i = 1,..., n do  for j = 1,...,i do  cij aijbjj  for k = j+1,..., i do  cij cij + aikbkj |
| Num of additions: =  =  = = = | | | | | | | Num of multiplications: = + =  + num of additions = + = | |
|  | = (n - 1) + 1 | | = = num of terms \* (1st term + last term) / 2 | | | | | | = |
| Upper Hessen-berg multi-plication | A = (aij)n x n be tridiagonal matrix = ,  B be upper triangular and  C = AB be upper Hessenberg matrix = (cij)n x n | | | | | 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  initialise cij = 0 for all i,j, where 1 ≤ j < i - 1 < n  for i = 1,..., n do  for j = max(1,i - 1),...,n do  k max(1, i - 1); cik aikbkj  for k = max(2, i),..., min(i + 1, j) do  cij cij + aikbkj | | | |
| Num of add:=  =  + =  1 + 2 + 2 + ... + 2 - n + =  1 + 2(n-1) - n + =  n - 1 + = n - 1 + n - 1 + 2n(n-1) - 2( - 1) = n2 - n | | | | | | Num of multiply: =  + num of add=  + num of add =  n(n+1) - [1 + 1 + 2 + 3 + ... + (n - 1)] + num of add =  n(n+1) - [1 + ] + n2 - n = (3n-2)(n+1) | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 into 2 until length of new interval/2 ≤ TOL. Soln: exact root r = approx root ± TOL (tolerance)  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)) | | | set a ao, b b0  while (b-a)/2 > TOL do  c (a+b)/2  if f(c) == 0: stop  else if f(a)f(c) < 0: b c  else a c  Result: appox root = (a+b)/2 |
| 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  However, due to rounding errors by computers, may terminate early and not have required digits of precision. | | | |
| Fixed-Point Iteration (FPI) | Solve eqn: g(x) = x. Keep applying g() to xi until num converge. r is a fixed pt if g(r) = r  Backward error = |g(xi+1) - xi+1| = |f(xa)|  Suppose f(x) = g(x) - x, and xa is approximation for r where f(r) = 0  Forward error = |r-xa|  Note that seq xi may not always converge.  Note any eqn f(x) = 0 can be turned into a fixed-pt problem g(x) = x | | set initial guess x0  for i = 0,1,2,... do (have max i in case divergent)  xi+1 g(xi)  if |g(xi+1) - xi+1| < TOL:  xfp g(xi+1), break  Result: xfp | |
| FPI con-vergence rate | Convergence Thrm: Suppose g is continuously differentiable, and g(r) = r and S := |g'(r)| < 1. Then FPI converges linearly with rate S to r for initial guesses sufficiently close to r  Let ei := |xi - r| denote error at step i. If = S < 1, mtd obey linear convergence with rate S  So S = 0: fastest convergence rate. S < 1: cfm converge. S > 1: won't converge | Fixed-Point Thrm: Let g be cts fn in [a,b] s.t. g(x) [a,b], x [a,b] and g' exists on (a,b) and constant 0 < k < 1 s.t. |g'(x)| ≤ k, x [a,b].  Then for any num xo in [a,b], seq xn = g(xn-1), n ≥ 1 converges to unique fixed pt r in [a,b] | | |
|  | Bisection Mtd: Guaranteed to converge linearly. However, need predefine initial interval and convergence rate = 1/2 (might be slower than FPI) | FPI: convergence rate S can be < 1/2, faster than bisection. But might be diff to formulate g from f | | |
| Horner's Mtd | 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)))  Horner's Mtd: most optimal mtd for finding value of polynomial. O(m)  pm = am. pm-1 = am-1 + xpm. ... p1 = a1 + xp2. p0 = a0 + xp1 | | P am  for k = m-1,...,0 do  P ak + xP  Result: P | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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: n(n-1)(2n+5)/6 (without labelling rows)  Multiplication: n(n-1)(2n+5)/6 (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 | | for i = 1,...,n: ri i (labelling rows) | | | | | | | | | |
| for i = 1,...,n-1: {  j i  while j ≤ n and = 0: j j+1 (check which col has pivot entry)  if j = n+1: Output: "Error: Singular matrix" (means have zero col)  else if j ≠ i: swap rj and ri | | | | | | | | | |
| for j = i+1,...,n: m­ji / (elimination: making matrix into upper triangular)  for k = i+1,...,n+1: (augmented matrix include bn x 1 as well)  - mji} | | | | | | | | | |
| xn /(backward substitution)  for i = n-1,...,1: {xi  for j = i+1,...,n: xi xi - xj  xi xi/} | | | | | | | | | |
| Partial Pivoting | | 1st mtd to perfrom row swap to minimise errors  Select pivot elem s.t. its absolute value is as large as possible in 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. | | | | | (labelling rows)  for i = 1,..., n-1 {l i  for j = i+1,...,n {if || > ||: l j}  if = 0: Output: "Error: Singular matrix"(row swap)  else if l ≠ i: swap rj and ri }  (elimination), (backward substitution) | | | | | | |
| Relative absolute ratio | | A way to do this is to compare the relative absolute sizes in each col j = 1,...,n ,  relative absolute size of row i = aij/max(row i)  By choosing largest relative absolute size for each col as pivot elem, loss of sig digits is minimised  Divisions = , Comparisons = | | | | | for i = 1,..., n-1  for j = i,..., n do (find max abs val in row)  ||  for k = i+1,..., n {if || > : ||}  l i, max ||/ (find max relative abs ratio)  for j = i+1,...,n {ratio ||/  if ratio > max: l j, max ratio} | | | | | | |
| Scaled Partial Pivoting (SPP) | | 2nd mtd to perform row swap to minimise errors  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 = | | | | | (labelling rows)  for i = 1,..., n {si |ai1|  for j = 2,..., n {if |aij| > si: si |aij|}  if si = 0: Output: "Error: Singular matrix"}  for i = 1,..., n-1 {(find max relative abs ratio), (row swap)}  (elimination), (back substitution) | | | | | | |
| 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, where L is lower triangular, U: upper triangular. LUx = b. L = strictly lower part of processed A + diagonal all 1. U = upper triangular of processed A | | | | | | | | | for i = 1,..., n-1 do (LU factorization)  for j = i+1,..., n do  aji aji/aii #store L in lower triangular part of A  for k = i+1,..., n {ajk ajk - ajiaik} #elimination | | |
| Forward substitution (solve Ly = b for y)  Backward substitution (solve Ux = y for x)  Time complexity for both substitution = O(n2)  So, applying Gaussian elimination p times = O(2pn3) | | | | y1 b1 (forward sub)  for j = 2,..., n do  yj bj  for i = 1,..., j-1: yj yj - Ljiyi | | | | | | xn yn/ann (backward sub)  for i = n-1,...,1 {xi yi  for j = i+1,..., n: xi xi - Uijxj  xi xi/Uii} | |
| PA = LU factoriza-tion  O(n3) + O(pn2) | | 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. Solve Ly = Pb, then solve Ux = y  No need to explicitly find P, just output r (stored row index) to replace P | | | Find max abs values of each row and initialise L, U, r  for i = 1,..., n-1 do (PA = LU factorization)  (find max relative abs ratio & row swap)  1, Uii  for j = i+1,..., n do  /  for k = i+1,..., n: -  Uij  1, Un,n | | | | | | | | (new forward sub)  y1  for j = 2,..., n do  yj  for i = 1,..., j-1:  yj yj - yi  backward sub still same as previous |
| 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) | | | | | | | | 1a. Apply row, col ops to introduce zeros into 1st col & row  1b. Apply row, col scaling so 1st diagonal entry becomes 1  2a. Apply row, col ops to introduce zeros into 2nd col & row  2b. Apply row, col scaling so 2nd diagonal entry becomes 1...  n. Apply row and col scaling so last diagonal entry is 1 | | | |
| Note also positive definite.  A = A1R1.  = A1. Since xTx = and A is symm positive definite, then = A1 also symm positive definite. Since is principal submatrix of A1, also symm positive definite  for k = 1,2,..., n do  if Akk < 0: Stop algo. A not positive definite  Rkk (diag entries of R)  vT ; Rk,k+1:n vT (rest of row k)  Ak+1:n,k+1:n Ak+1:n, k+1:n - vvT (principal submatrix) | | | | | | A = = = = A1R1 where u1 = , K1 = . Then repeat finding for and so on...Finally will get RT = | | | | | |
| Jacobi Method  O(n2) | | 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, =  = ...  =  So calculation can be parallelized | | | | | | | = + + = L + D + U | | | | |
| set initial vector x(0)  d diag(A); R A - diag(d) (to get L + U)  for k = 0,1,... do  x(k+1) = (b - Rx(k))d ( = element-wise division) | | Terminate algo when max iteration is reached  OR x(k+1) approximately solve L.S, i.e. max1 ≤ i ≤ n{|Ax(k+1) - b|i} <  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 = Jacobi mtd converge  If A sdd, A is nonsingular matrix. If A not sdd, MIGHT still converge | | | | | | | | | |
|  | | 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. | | | | | | | | | | | |
| Gauss-Seidel Method O(n2) | | Similar to Jacobi. Now, x(k+1) = D-1(b - Lx(k+1) - Ux(k))  = (no change)  = ...  =  Now cannot parallelized, as x2,... xn dependent on x1, ..., xn-1  But since updated values are used, will converge faster than Jacobi  If A is sdd, Gauss-Seidel mtd will converge | | | | | | set initial vector x(0)  for k = 0,1,2,... do  for i = 1,2,...n do  bi  for j = i+1,..., n: { - }  for j = 1,..., i-1: { - } | | | | | |
| Succes-sive Over-Relaxation (SOR) Method  O(n2) | | Gauss-Seidel iterate: = D-1(b - Lx(k+1) - Ux(k))  Current iterate: x(k)  Let be a real num, and define x(k+1) as weighted average of and x(k). i.e. x(k+1) = (1 - )x(k) +  is called relaxation parameter and >1 = over-relaxation  GS: = 1. SOR: > 1  Faster convergence than GS  Need to choose wisely, usually 1.1 or 1.2 | | | | | | set initial vector x(0)  for k = 0,1,2,... do  for i = 1,2,...n do  bi  for j = i+1,..., n: { - }  for j = 1,..., i-1: { - }  (1 - ) + | | | | | |
| 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 fn g to connect these pts to restore original fn f  Interpolation: restoration of f  Interpolant / Interpolating fn: g  Interpolation nodes: x0, x1, ..., xn | Properties of g: Simplicity (easy to evaluate) and approximability (accuracy)  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]  If we have n+1 data points, polynomial must be of degree n, with n+1 coeff  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 | | | | g must satisfy: g(x0) = f(x0), g(x1) = f(x1), ..., g(xn) = f(xn)  Using basis fn, g(x) = f(x0)(x) + f(x1)(x) + ... + f(xn)(x)  E.g. g(x0) = f(x0)(1) + f(x1)(0) + ... + f(xn)(0) = f(x0) | |
| Lagrange basis poly-nomial | Let x0, x1, ..., xn be the n+1 distinct real nums. For k = 0, 1, ..., n, the  kth Lagrange basis polynomial Lk(x) is a polynomial of degree n where Lk(x) = = . So Lk(xj) = | | | | | Then interpolating fn, P(x) = f(x0)L0(x) + f(x1)L1(x) + ... + f(xn)Ln(x) which satisfies P(x0) = f(x0), P(x1) = f(x1), ..., P(xn) = f(xn)  P(x) is called the Lagrange interpolating polynomial |
|  | 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 | |
| Unique-ness of Lagrange poly-nomial | 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) = f(x0)L0(x) + f(x1)L1(x) + ... + f(xn)Ln(x) where Lk is the kth Lagrange basis polynomial | | | | Suppose there is another polynomial having deg ≤ n satisfying (xk) = f(xk), for k = 0,1,...,n. Then (xk) - Pn(xk) = 0, for k = 0,1,...,n  This means xk, k = 0,1,...,n are roots of polynomial (xk) - Pn(xk) and has at least n+1 roots. But and Pn are deg ≤ n.  By contradiction, Hence = Pn | |
| 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)).  Pn-1(x) = f(x0)L0(x) + ... + f(xn-1)Ln-1(x) where Lk(x) =  Naive way: Recalculate Lagrange basis polynomial again  Pn(x) = f(x0)L0(x) + ... + f(xn)Ln(x) where Lk(x) = | | | 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 (Proof in l13)  nth divided diff of f = f[x0, x1, ..., xn] =  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) | | |
| 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] (proof in t8) | | | Let x0, x1, ..., xn be n+1 distinct real nums. Then f[x0, x1, ..., xn] = . (Proof in l14)  Compute f[x0], f[x1]...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 | | |
| Divided Diff & Computa-tion | for k = 0,1,...,n do {f[xk] f(xk)}  for j = 1,...,n do {  for k = 0,...,n-j do {f[xk,...,xk+j] (f[xk+1,...,xk+j] – f[xk,...,xk+j-1])/(xk+j – xk)}  } Result: all f[xj,...,xk] for 0 ≤ j ≤ k ≤ n | | | | | 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 fn: f(x) = , x [-1, 1]  Runge's phenomenon: Wider oscillation at ends near interpolation interval (worse interpolation at ends). num of nodes only worsen approximation at ends of interval.  Error of interpolation, |f(x) - Pn(x)| : error is 0 on all nodes, error has peak btw every pair of adjacent nodes, peaks closer to 1st and last nodes are higher than nodes in the middle  gn(x) = |(x-x0)(x-x1)...(x-xn)| = has similar pattern to error |f(x) - Pn(x)| w equally-spaced nodes  = 0 as Pn only deg ≤ n.  (x-x0)(x-x1)...(x-xn)] = (n+1)! since any x w deg < n+1, after differentiating n+1 times would become 0 | | 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 , there is a depends on x0, x1, ..., xn, x, and lies btw the min and max of {x0, x1, ..., xn, x}, so that f(x) - Pn(x) = (x-x0)(x-x1)...(x-xn) = Tn+1(x)  Proof. For x {x0, x1, ..., xn}, LHS = f(x) - Pn(x) = f(x) - f(x) = 0 = RHS  For x {x0, x1, ..., xn}. Let h(x) = f(x) - Pn(x) - (x-x0)(x-x1)...(x-xn).  For h(x) = 0, now prove there exist s.t. =  Since h(x0) = h(x1) = ... = h(xn) = h(x) = 0, by IVT, min(x, x0) < y0 < ... < yn < max(x, xn), s.t. h(1)(yi) = 0, i = 0,1,...,n  Continue applying IVT until h(n+1)() = 0. Also, h(n+1)() = f(n+1)() - () - (x-x0)(x-x1)...(x-xn)] = f(n+1)() - (n+1)!. So = | | | |
| Cheby-shev Interpola-tion | Chebyshev nodes: xk = cos, k = 0,1,..., n in [-1, 1]  Using Chebyshev nodes means max value of gn(x) is the smallest, with min = . Min is achieved by = Tn+1(x), where Tn+1(x) denotes the deg n+1 Chebyshev polynomial = cos((n+1)arccos x)  (Proof in L15)  So now gn(x) = | cos((n+1)arccos x)| ≤ . gn 0 as n ∞  For [a,b]: xk = + cos, k = 0,1,..., n  For [a,b]: ≤ | | | | Now, all peaks have same height (error is more evenly distributed). Difference btw nodes is smaller near bounderies  By using Chebyshev nodes to interpolate w divided diff, this polynomial = Chebyshev interpolating polynomial  Approximation using Chebyshev nodes is worse ard center, but oscillation near ends much milder. As n increase, Chebyshev interpolating polynomial will converge to f(x) | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Linear Least Square Problem | If use interpolation, deg of polynomial is higher and data points might contain errors.  Inconsistent sys: SLE w no solution, typically m > n  So find s.t. A is closest to b, A lies on plane Ax  b - A is perpendicular to plane.  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  So, (b-A) {Ax|x}. (Ax)T(b-A) = 0. xTAT(b-A) = 0.Since x ≠ 0, AT(b-A) = 0. ATA = ATb. is L.S sol to Ax = b, which minimizes Euclidean norm 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 factoriza-tion | ATA not numerically stable as could have rounding errors.  So, use (reduced) QR factorization: A = QR, where Q is an orthogonal matrix (QT = Q-1), R is upper triangular. Then ATAx = ATb, (QR)T(QR)x = (QR)Tb, RTRx = RTQTb, Rx = QTb if RT is invertible.  So given **a**1, **a**2, ..., **a**n (m ≥ n), need to find  **q**1, **q**2, ..., **q**n  s.t. they are unit vectors (i.e. ||**q**i|| = 1) and **q**i, **q**j are orthogonal to each other when i ≠ j  r­11, r12, r22, ..., rnn | | =  = = | | |
| Gram-Schmidt orthogo-nalization | **y**1 = **a**1, **q**1 =  **y**2 = **a**2 - r12**q**1, **q**2 =  **y**j = **a**j - r1j**q**1 - r2j**q**2 - ... - rj-1,j**q**j-1, **q**j =  rjj = ||yj||, rij = **a**j | | for j = 1,2,...,n do  y **a**j  for i = 1,2,..., j-1 do  rij **a**j, **y** **y** - rij**q**i  rjj ||**y**||, **q**j **y**/rjj | | |
|  | Reduced QR factorization: m x n = (m x n) \* (n x n)  = | | | Full QR factorization: m x n = (m x m) \* (m x n)  = | |
| QR steps | Compute reduced QR factorization of A. Compute QTb. Solve Rx = QTb using backward sub | | | | |

­­­­

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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(a) + f'(a)(x-a) + f''(a)(x-a)2 becomes f(x+h) = f(x) + hf'(x) + h2f''(a)  OR f'(x) ≈ with error f''(c) where c [x,x+h] | | Error f''(c) is O(h) (proof in l18), i.e. if h decr by , error also decr by . Since error is O(h), 2-points forward-diff formula is a 1st-order method  If error is O(hk), then formula is a k-order approximation | |
| 2nd order finite diff formula | | If f is 3 times cts diff-tiable, then f(x+h) = f(x) + hf'(x) + f''(x) + f'''(c1) & f(x-h) = f(x) - hf'(x) + f''(x) - f'''(c2) where x-h < c2 < x < c1 < x+h  So f(x+h) - f(x-h) = 2hf'(x) + h3f'''(c1) + h3f'''(c2), then f'(x) = - [h2f'''(c1) + h2f'''(c2)] = approx - error | | | |
| Genera-lized IVT | | Let f be a cts fn on interval [a,b]. Let x1, ..., xn be points in [a,b] and a1, ..., an > 0. Then c [a,b] s.t.  (a1 + ... + an)f(c) = a1f(x1) + ... + anf(xn) | Proof. Let f(xi) = min and f(xj) = max of n fn values  a1f(xi) + ... + anf(xi) ≤ a1f(x1) + ... + anf(xn) ≤ a1f(xj) + ... + anf(xj). f(xi) ≤ ≤ f(xj)  By IVT, c [xi,xj] s.t. f(c) = OR (a1 + ... + an)f(c) = a1f(x1) + ... + anf(xn) | | |
| 3-point centered-diff formula | | f'(x) = - [h2f'''(c1) + h2f'''(c2)]. Using Generalized IVT, h2f'''(c1) + h2f'''(c2) = a1g(c1) + a2g(c2) = (a1+a2)g(c) = h2f'''(c), where x-h < c < x+h  So, f'(x) ≈ w error h2f'''(c), where x-h < c < x+h | | | Since error is h2f'''(c) = 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  f(x+h) + f(x-h) = 2f(x) + h2f''(x) + h4f(4)(c1) + h4f(4)(c2)  3-point centered diff formula for f''(x) ≈ w error f(4)(c) where x-h < c < x+h | | | |
| Rounding error | | When 2 nums nearly equal, loss of sig digits due to computer rounding error. Suppose (x+h), (x-h) are 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 formula = |f'(x) - '(x)| = = ≤ ≤ , where , < > 0 and |f'''(c)| ≤ M. So |f'(x) - '(x)| ≤ . Smallest error at h = | | | |
| Extra-polation | | order n formula for approximating Q: Q = Fn(h) + K(h)hn  For 3-point centered formula, F2(h) = , K(h) = . If f''' is cts and h is not large, then values of – should roughly be constant in a small interval containing c  Q - Fn(h/2) = K(h/2)(h/2)n = K(h/2)hn ≈ K(h)hn = (Q - Fn(h)). Then Q ≈ : (Richardson) extrapolation formula for Fn(h)  Using Taylor expansion at point 0, K(h) = K(0) + O(h). Q = Fn(h) + K(h)hn = Fn(h) + (K(0) + O(h))hn = Fn(h) + Bhn + O(hn+1), where B = K(0)  Then Q = Fn(h/2) + B(h/2)n + O(hn+1), implying = ... = Q + O(hn+1). Thus Q = + O(hn+1) := Fn+1(h) + O(hn+1)  Using extrapolation, appox for Q is of higher accuracy as Fn+1(h) is at least an order n+1 formula, compared to original Fn(h) | | | |
| E.g. | | Using f'(x) = . = ... = = F(h), a five-point centered-diff formula  By observation, since F(h) = F(-h), error term can be even powers of h only. Since original order is 2, 2+1 = 3. New order ≥ 3, thus must be 4 | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Newton-Cotes approach | Use definite integral of the interpolating polynomial of f to approximate  Trapezoid Rule: Use Lagrange interpolating polynomial. Let y0 = f(x0), y1 = f(x1)  f(x) = + f''(cx) := P1(x) + E(x) for some cx depending on x  = ... = (y0 + y1) - f''(c) := approx + error where h = x1 - x0 and c [x0, x1] | | Proof in lect 20.  Uses MVT for integrals: Let f be cts fn on interval [a,b] and let g be an integrable fn that does not change sign on [a,b]. Then c [a,b] s.t. = f(c) |
| Simpson's Rule: Use Lagrange interpolating polynomial. y0 = f(x0), y1 = f(x1), y2 = f(x2)  f(x) = + f'''(cx) := P2(x) + E(x) for some cx depending on x.  = ... = (y0 + 4y1 + y2) - f(4)(c) := approx + error where h = x1 - x0 = x2 - x1 and c [x0, x2] | | Assume nodes evely spaced. Proof in lect 20.  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. Proof in tut? |
| Composite Newton-Cotes Formulas | Trapezoid and Simpson's Rule only operating on single interval  Composite numerical integration: Divide interval into several subintervals  Composite Trapezoid Rule: = (y0 + 2 + ym) - f''(c) where h = (b-a)/m and c [a,b]. Order of comp trapezoid rule = 2, O(h2). Proof in lect 20. | | Composite Simpson's Rule: = (y0 + 4 + 2 + y2m) - f(4)(c) := approx + error where h = (b-a)/(2m) and c [a,b]  Order of comp Simpson's rule: 4. O(h4) |
| 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]. Proof in lect 21. | | Composite Midpoint Rule: = h + f''(c). O(h2) |
| Romberg integration | Composite Trapezoid Rule: = (y0 + 2 + ym) - f''(c)  Using Richardson extrapolation would give us a new rule of at least 3rd order  For infinitely differentiable fn f, = (y0 + 2 + ym) +c2h2 + c4h4 + c6h6 + …, where ci depends only on higher derivatives of f at a and b and not h (e.g. c2 = (f'(a) - f'(b))  Let Q := , F2(h) := = (y0 + 2 + ym). Then Q = F2(h) + c2h2 + c4h4 + c6h6 + … | | Cutting h in half and combining, F4(h) := . Q = F4(h) + h4 + h6 + … = F4(h) + O(h4). Proof in lect 21  Cutting h in half again and combining, F6(h) := . Q = F6(h) + h6 + … = F6(h) + O(h6). Proof in lect 21  So F2k(h) := |
| Text, whiteboard  Description automatically generated | Text  Description automatically generated with low confidenceRomberg triangle: | |
| Not tested | Gaussian Quadrature, Legendre polynomials | | |