Lecture Notes Numerical Methods

Mick Veldhuis

April 8, 2020

Contents

1 Random numbers

First, note that random number generators are never truly random, but rather pseudo-random as computers are deterministic. By definition a good RN generator is one such that the generated numbers are uncorrelated with each other, drawn from the specified distribution and that the *periodicity* (the number of steps after which the generator will start repeating itself) is much greater than the number of generated data points.

1.1 Uniform deviates

Uniform deviates are random numbers drawn from a range of equal probability, i.e. $x \in (0, 1)$ such that all x have equal probability. A generator is a function that, given the same *seed*, returns the same sequence of random variates.

1.2 Linear congruential generators

To generate a sequence $\{x\}$ for $x \in \mathbb{Z}$ on the interval $[0, m)$ (where m is some large number), by sequentially applying

$$I_{j+1} = aI_j + c \pmod m \quad \Rightarrow \quad x_{j+1} = I_{j+1}/m$$

where a and c are resp. the multiplier and the increment (choose a and c wisely and the periodicity might be $\leq m$). Note however that there exist sequential correlations between successive iterations. This was also demonstrated by *Marsaglia*, who demonstrated that if one generated points in this matter they will not be fully independent.

1.3 Multiplicative congruential generators

To generate a sequence $\{x\}$ for $x \in \mathbb{Z}$ on the interval $[0, m)$ (where m is some large number), sequentially apply

$$I_{j+1} = aI_j \pmod m \text{ with } I_0 \neq 0 \quad \Rightarrow \quad x_{j+1} = I_{j+1}/m$$

Again a and m have to be chosen wisely, e.g. $a = 7^5$ and $m = 2^{31} - 1$. Note: in 32-bit systems, this routine will exceed the maximum integeter $= a(m - 1)$.

Often it's better to use more complex generators, as correlations exist between different sequences. One could for instance combine sequences with different periods (i.e. P_1, P_2), which results in sequences with period $P_1 \times P_2$. This can be achieved by reshuffling the output sequences in some *random* manner.

1.4 Transformation method

Assume that we know how to generate a random number via a uniform distribution $f(x)\,\mathrm{d}x = 1\,\mathrm{d}x$ for $0 \leq x < 1$ and suppose we have a function $y(x)$ and we want its probability distribution $p(y)$, we could do the following to obtain $p(y)$:

$$p(y)\,\mathrm{d}y = f(x)\,\mathrm{d}x \quad \Rightarrow \quad p(y) = f(x)\left|\frac{\mathrm{d}x}{\mathrm{d}y}\right|$$

But what if we know $p(y)$ and want to generate random deviates y? We do the following, assuming that x is uniform such that we have $p(y)\,\mathrm{d}y = 1\,\mathrm{d}x$:

1. Calculate the CDF: $x = F(y) = \int p(y)\,\mathrm{d}y$
2. Calculate the inverse CDF: $y = F^{-1}(x)$, again note that x is drawn from a uniform distribution

1.5 Rejection method

A more powerful method than the transformation method, because it is more general (we don't need the CDF/inverse CDF). Essentially we compare $p(x)$ to some other distribution $f(x)$ (the CDF of $f(x)$ should be easy to determine), where $f(x) \geq p(x)$. To draw values according to $p(x)$ we use the following recipe:

(1) Draw a value y_0 uniformly in $(0, A)$, where $A = \int f(x)\,\mathrm{d}x$.

(2) Compute $x_0 = F^{-1}(y_0)$, where $F^{-1}(y)$ is the inverse CDF of $f(x)$.

(3) Draw y_1 uniformly in $(0, f(x_0))$.

(4) If $y_1 \leq p(x_0)$ the value x_0 is recognised as being drawn from $p(x)$, otherwise discard x_0.

(5) Repeat from (1) until we have the desired number of random deviates.

2 Linear algebra

Imagine we have a systems of equations $A\mathbf{x} = \mathbf{b}$, where $A \in \mathbb{R}^{m\times n}$; n is the number of unknowns and m the number of equations. If $m = n$ there may be a unique solution, if $\det(A) = 0$ then the system is is singular (i.e. has no solution).

(1) If $m < n$ or $\det(A) = 0$ the system is underdetermined and the may be no unique/multiple solutions. In this case: if there are multiple solutions, the solution space consists of a linear combination of vectors of a particular solution. Some of the eigenvalues of A are zero. For one eigenvalue the solution space is a line, for two a plane, etc.

(2) If $m > n$ the system is overdetermined. There is no unique solution, but one can find a solution in the least squares sense.

2.1 Methods for solving linear algebraic equations

2.1.1 Gauss-Jordan elimination

The idea is to perform row/column operations to reduce A to I, e.g. by interchanging two rows, multiplying by a non-zero number, or adding a multiple of one row to another. But this method is hardly ideal:

(1) Diagonal elements can become zero.

(2) G-J elimination is unstable to round-off errors. This issue can be combated using *pivoting*, meaning the largest elements are put on the diagonal by rearranging rows and columns.

(3) Memory requirements.

2.1.2 LU decomposition

The idea is to write the matrix A as $A = LU$, where L is a lower triangular matrix and U an upper triangular matrix. Such that we can now solve the system as $A\mathbf{x} = (LU)\mathbf{x} = L(U\mathbf{x}) = \mathbf{b}$, we could then first solve $L\mathbf{y} = \mathbf{b}$ and then $U\mathbf{x} = \mathbf{y}$.

Advantages are: (1) the number of operations $< n^3$, (2) once the LU decomposition is found it is trivial to find the solution for different $\mathbf{b}$. Note: using L and U we can also easily find the inverse and the determinant of A, i.e. the determinant of A: $\det(A) = (-1)^S \det(L)\det(U)$, where S is the number of permutations made to obtain the decomposition. (Be wary though, computation of the determinant in this manner is prone to over/underflowing the dynamic range = ratio between the largest and smallest values that a certain quantity can assume)

2.1.3 Singular value decomposition: SVD

This method works very well in case of an almost singular matrix and can be used to solve LLS problems. This method is based on the fact that a matrix A with $m \geq n$ can be written as

$$A = UWV^T$$

where

(1) $U \in \mathbb{R}^{m\times n}$ and column orthogonal

(2) $W \in \mathbb{R}^{n\times n}$ and diagonal (consisting of singular values: positive or zero).

(3) $V \in \mathbb{R}^{n\times n}$ and orthogonal, so $V^T = V^{-1}$.

If A has non-zero singular values: (1) the columns of U are (a subset of) the eigenvectors of AA^T, (2) the columns of V are the eigenvectors of A^TA, (3) the singular values w_j are $\sqrt{\lambda}$, where λ are the eigenvalues of A^TA, (4) the matrix W is unique, however U and V don't have to be.

SVD of a square matrix: In this case $A^{-1} = VW^{-1}U^T$ with $W^{-1} = \text{diag}(1/w_j)$. To check whether A is invertible we compute the **condition number** = $\max(w_j)/\min(w_j)$. If the condition number is infinite the matrix is singular, if the condition number is very large the matrix is ill-conditioned.

Nullspace and range: The nullspace is the set of vectors $\mathbf{x}$ such that $A\mathbf{x} = \mathbf{0}$, which has dimensions (nullity) equal to the number linearly independent vectors $\mathbf{x}$. The range of A is the subspace of $\mathbf{b}$ where $A\mathbf{x} = \mathbf{b}$ is possible.

Note that SVD constructs an orthogonal basis for the nullspace and the range of a matrix: (1) the columns of U corresponding to non-zero w_j are orthogonal basis of the range, (2) the columns of V corresponding to $w_j = 0$ are orthogonal basis of nullspace.

2.1.4 Least squares solution

If a system of equations does not have a unique solution and the number of unknowns is much smaller than the number of data points we try to minimize $G = (A\mathbf{x} - \mathbf{b})^T(A\mathbf{x} - \mathbf{b})$ as $\mathrm{d}G\,/\,\mathrm{d}x = 0$. In the end we find that the least squares solution is given by

$$A^T A\mathbf{x} = A^T\mathbf{b}$$

Another way might be via the pseudo inverse, which is essentially the inverse found by applying SVD to A. In that case it is important that A is not ill-conditioned; such that $1/w_j >> 1$. To counter this problem one should set $w_j = 0$.

3 Interpolation

Interpolation involves fitting a function (i.e. $f(x)$, often polynomials) to the data points at some values of x and then we can evaluate the function for any x. The order of the interpolation is given by the degree of the polynomial used and is typically also related to the number of points used per interval. Note that it is often best to interpolate locally (i.e. using only a subset of the points) as points closer to each other are more correlated.

3.1 Splines

Splines[1] are essentially polynomials, but the information used to construct them is less local. Splines link the polynomials across nodes via their derivatives (so having continuous derivatives is important) at the connecting nodes. In general if one has N points one can find the polynomial through those points of degree $N - 1$, that polynomial is given by *Lagrange's formula.*

3.1.1 Linear splines

When fitting a linear function, we only have two free parameters. So we have two linear equations with two unknowns, thus one solution. Note that, unless the actual function is linear, there is no continuity in the derivative at the end points of each sub-interval.

3.1.2 Quadratic splines

Using quadratic splines[2] we have three free parameters, and two options/sub-intervals. We specify three points where the function and polynomial should coincide and specify two boundary conditions such that we have continuous derivatives at the edges of each sub-interval. On each interval we define a polynomial:

$$p_i(x) = a_i + b_i x + c_i x^2 \text{ for } x \in [x_i, x_{i+1}],\ i = 1, \ldots, N$$

with initial/boundary conditions

[1]For a more detailed account of splines: `https://www.math.uh.edu/~jingqiu/math4364/spline.pdf`

[2]Interesting read about the use of quadratic splines: `https://wordsandbuttons.online/quadratic_splines_are_useful_too.html`

$$p_i(x_i) = y_i,\ i = 1, \dots, N+1$$

$$p_i(x_{i+1}) = p_{i+1}(x_{i+1}),\ i = 1, \dots, N-1$$

$$p_i'(x_{i+1}) = p_{i+1}'(x_{i+1}),\ i = 1, \dots, N-1$$

Thus we have $3N$ unknows (3 per polymial) and therefore we have $3N-1$ initial/boundary conditions. To actually fully solve the system one would need an additional constraint, for instance that the first derivative be zero ($p_1'(x_1) = 0$).

3.1.3 Cubic splines

The goal of the cubic spline is to be continuous in the first derivative and continuous in the second derivative within the interval and at the boundaries. Such that we now have the following initial/boundary conditions

$$p_i(x_i) = y_i,\ \text{and } p_N(x_{N+1}) = y_{N+1},\ i = 1, \dots, N$$

$$p_i(x_{i+1}) = p_{i+1}(x_{i+1}),\ i = 1, \dots, N-1$$

$$p_i'(x_{i+1}) = p_{i+1}'(x_{i+1}),\ i = 1, \dots, N-1$$

$$p_i''(x_{i+1}) = p_{i+1}''(x_{i+1}),\ i = 1, \dots, N-1$$

For each polynomial

$$p_i(x) = a_i + b_i x + c_i x^2 + d_i x^3 \text{ for } x \in [x_i, x_{i+1}],\ i = 1, \dots, N$$

The total number of equations is $4N-2$ ($3N-3$ for continuity and $N+1$ from interpolation), therefore we need two additional constraints. One could for instance set the second derivatives zero (natural cubic spline) at both edges or if the values at the edges is known one could use those, or put conditions on the first derivatives (complete cubic spline).

3.2 2D interpolation

We assume that the function is tabulated on a regular Cartesian grid, we would like to get the value of y at (x_1, x_2). If we want the value within one of those squares in the grid we define $y_1, \dots, y_4$ at all the nodes. The concept[3] is a combination of linear interpolation along x_1 and then along x_2. Like with 1D interpolation, the first derivatives are not continuous on the grid points.

Improvements: Go to higher orders or enforce smoothness (bicubic splines). When using bicubic splines one would also have to specify the gradients at the grid points.

4 Integration

The goal is to find a solution – as accurately as possible with the least amount of evaluation – to

[3]See page 4 for a quick run down of the algorithm behind bilinear interpolation: `https://www.cs.umd.edu/~djacobs/CMSC427/Interpolation.pdf`

$$I = \int_a^b f(x)\,\mathrm{d}x$$

In general one could evaluate the function at $x_i = x_0 + ih$ (for $i = 1, \dots, N$) and sum the result.

4.1 The trapezoidal rule

4.1.1 The basic rule

On an interval (x_1, x_2) we can approximate the integral as

$$\int_{x_1}^{x_2} f(x)\,\mathrm{d}x = \tfrac{h}{2}\left[f(x_1) + f(x_2)\right]$$

This can then be repeated N times to obtain

$$\int_{x_1}^{x_N} f(x)\,\mathrm{d}x = \frac{h}{2}\left[f(x_1) + \sum_{i=2}^{N-1} f(x_i) + f(x_N)\right]$$

The larger N, the smaller the error.

4.1.2 On an open interval

Suppose you'd want to calculate the integral

$$\int_{x_0}^{x_1} f(x)\,\mathrm{d}x$$

and you don't want to calculate the value at x_0. Then the first order approximation is

$$\int_{x_0}^{x_1} f(x)\,\mathrm{d}x = hf(x_1)$$

On the whole open interval (x_1, x_N) we would obtain

$$\int_{x_1}^{x_N} f(x)\,\mathrm{d}x = h\left[\frac{3}{2}f(x_2) + \sum_{i=3}^{N-1} f(x_i) + \frac{3}{2}f(x_{N-1})\right]$$

We could imagine an algorithm in which you would sequentially add the evaluation for $N = 1, 2, \dots$ until a certain moment, say when the difference between two evaluations is less than a certain tolerance:

$$\Delta I = I(i+1) - I(i) \leq \varepsilon$$

If one would implement an algorithm with first using N steps and then $2N$ steps we obtain *Simpson's rule* and we define

$$S = \frac{4}{3}S_{2N} - \frac{1}{3}S_N$$

such that the leading error term will cancel and the next term will be of order $1/N^4$. If one makes further refinements to get an error of order $1/N^{2k}$ we need k refinements (this is known as *Romberg* integration).

4.2 Dealing with improper integrals

1. The integrand has a limit at the upper and lower limits of the integral but cannot be evaluated there. For this we require an algorithm to work on an open formula, i.e. the **midpoint-rule** where the function is evaluated at the midpoints of the interval.
2. Upper or lower limits are $\pm\infty$. In this case one could do a change of variables (i.e. $x \to 1/t$, such that $ab > 0$)
3. It has an integrable singularity at either limit or it has an integrable singularity at a known place between the limits. If the singularities are of power-law type, one can perform a change of variables. I.e. if the divergence is $1/(x-a)^\gamma$, with $\gamma \in [0,1)$, we define $t = (x-a)^{1-\gamma}$.
4. It has an integrable singularity at an unknown place in the interval.

5 ODEs

Problems involving ODEs imply finding the solution $y(x)$, in e.g.

$$y''(x) + q(x)y'(x) = r(x)$$

which can be converted to the following system of first order derivatives:

$$y'(x) = z(x) \text{ and } z'(x) = r(x) - q(x)z(x)$$

Note: always try to keep the function(s) smooth and well-behaved. In general we require boundary conditions to solve a(n) (system of coupled) ODE(s). In that category we have initial value problems and boundary value problems. All methods to solve ODEs rely on *finite-differences*, i.e. $\lim_{\Delta t\to 0} \Delta y/\Delta t = \mathrm{d}y\,/\,\mathrm{d}t = f(t)$, resulting in $\Delta y \approx f(t)\Delta t$. Common methods are Runge-Kutta (robust, but not very fast) and leapfrog integration (rather specific method, time-reversible and conserves the energy of a system).

5.1 Euler's method

A rather bad method, but conceptually important.

$$y_{n+1} = y_n + hf(x_n, y_n), \quad f(x_n, y_n) = y'(x_n) \text{ and } h = x_{n+1} - x_n$$

In each iteration, only info used is value of fat beginning of interval, thus the method is not symmetric! The error on the estimate is of order h^2.

5.2 Runge-Kutta

A second order method with errors of order h^3. It uses info at the beginning and end of the interval to make a more accurate prediction. We define

$$k_1 = f(x_n, y_n) \text{ and } k_2 = f(x_n + h/2, y_n + \tfrac{h}{2}k_1)$$

such that $y_{n+1} = y_n + hk_2$. We can also extend the method to a 4th order variant in which we evaluate the derivatives at more points in the interval to increase the accuracy. Essentially the derivative is evaluated once at initial point, twice at trial midpoints, and once at trial endpoint. From these the final value is calculated via a weighted average (to cancel lower order errors). This is cost-effective if the step is relatively large. Now we have an error of order h^5.

5.3 Dormand-Prince method

This method is based on RK4, but calculates seven different slopes. These slopes are then used in two different linear combinations to find two approximations of the next point. One is of other h^4, the other of h^5. These two approximations can then we used to find the adaptive step size.

5.4 Leapfrog integration

Suppose we want to solve an ODE of the form

$$\frac{\mathrm{d}^2 x}{\mathrm{d}t^2} = F(x)$$

The necessary steps are

$$x_{i+1} = x_i + hv_{i+1/2}$$

$$v_{i+3/2} = v_{i+1/2} + hF(x_{i+1})$$

Since our initial values for position and velocity are at the same instant in time we need to jump start the velocity by using

$$v_{1/2} = v_0 + \tfrac{1}{2}hF(x_0)$$

Finally we also would like the ability to re-synchronize the velocities with the positions with respect to time. Such that we can actually use the velocity in calculating the total energy and angular momentum of the system. This is done by iterating over the positions and velocities and evaluating

$$v_{i+1} = v_{i+1/2} + \tfrac{1}{2}hF(x_{i+1})$$

6 Root finding

Root finding consists of finding x such that $f(x) = 0$ (or in higher dimensions $\mathbf{x} = [x_1\ x_2\ \ldots]$ such that $f(\mathbf{x}) = 0$). Most algorithms work by iterations from an initial guess, here it is extremely important that you pick good initial guesses, for instance via inspection. Typically one would have to bracket the function between two values $[a, b]$, with $\operatorname{sign} f(a) \neq \operatorname{sign} f(b)$. The **intermediate value theorem** states that if the function is continuous on (a, b), then the root must lie within the interval.

6.1 Bisection method

Based on this simple recipe:

1. Check that the function is properly bracketed.
2. Evaluate the function at the midpoint and examine the sign.
3. Replace the interval with the midpoint depending on where the function has the same sign as the midpoint. I.e. if $f(a)f(x_{\text{mid}}) > 0$, we set $a = x_{\text{mid}}$.

After i iterations the root is known within an interval of size ε_i, such that the following iteration $\varepsilon_{i+1} = \varepsilon_i/2$ (so linear convergence). The number of iterations n needed to converge is

$$n = \log_2\left(\frac{|a-b|}{\varepsilon}\right)$$

where ε is desired tolerance. Often it is best to define a relative tolerance and scale to the machine precision ε_{acc}, $\varepsilon = \varepsilon_{\text{acc}}(|a|+|b|)/2$.

6.2 Secant method

Good method for functions that are smooth near the root, as this method converges more quickly than the bisection method. We assume that the function is approximately linear in the region of interest. Each improvement is taken as the point where the approximating line crosses the axis. The secant method retains only the most recent estimate, so the root does not necessarily remain bracketed (i.e. local behaviourcan send the iteration to infinity).

$$x_{n+1} = x_n - \frac{(x_{n-1}-x_n)f(x_n)}{f(x_{n-1})-f(x_n)}$$

6.3 Newton-Raphson method

6.3.1 The 1D case

The idea of this method is to determine the tangent defined at one point, find where it crosses zero, and taking that point as the next guess. So every iteration of the algorithm we calculate

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

and then check whether

$$\frac{|x_{n+1}-x_n|}{|x_{n+1}|} \leq \text{rtol}$$

If that condition is met, we have the root, i.e. x_{n+1}. Note: problems appear if one of the iterations lies close to a minimum or maximum, as that will send the next point to limbo. The solution is to use bracketing first. The Newton-Raphson method converges quadratically. This method is preferred for any function whose derivative is known (analytically)/easily evaluated (and of course whose derivative is continuous and nonzero in the neighborhood of a root).

6.3.2 Extending it to multiple dimensions

Suppose we seek the solution to

$$\begin{aligned} f_1(\mathbf{x}) &= 0 \\ f_2(\mathbf{x}) &= 0 \\ &\vdots \\ f_n(\mathbf{x}) &= 0 \end{aligned}$$

In 2D, for two arbitrary function, each will have a contour in the (x, y) plane that satisfies each equation, but these contours will not necessarily cross,meaning a solution may not exist.

If we want to extend the Newton-Raphson method to multiple dimensions, instead of the derivative we now require the Jacobian J. Then to determine the roots of the system we calculate $J\delta\mathrm{x} = -F(\mathrm{x})$ every iteration and solve for $\delta\mathrm{x}$ using LU decomposition and calculate $\mathrm{x}_{new} = \mathrm{x}_{old} + \delta\mathrm{x}$.

We'd also need to define a tolerance, i.e. w.r.t. the function $\|f\| < \mathrm{tol}_f$ or the root $\|\delta\mathrm{x}\| < \mathrm{tol}_{\mathrm{x}}$. If such a condition is met we terminate the process and we have subsequently found the position of the root, i.e. x_{new}.

7 Minimization & maximization

In general we want to find the global minimum of a function. To do so we use either solely the function or also its first derivate(s)/gradient.

7.1 Golden section search

Similar idea to the bisection method. In case of looking for minima we bracket the minimum via $X < Y < Z$, such that $f(Y) < f(X)$ and $f(Y) < f(Z)$. The central point of the triplet is always the best guess for the minimum. There are optimal ways of choosing next point after bracketing, and we continue until the distance between X and Z is sufficiently small. Note that in comparison to root finding, finding the minimum is generally less accurate.

7.2 Brent's method

The idea behind this method is that you do a quadratic interpolation through the points a, b, c that satisfy $f(a), f(c) > f(b)$. So you make trial parabola through the 3 bracketing points, then jump to the minimum of this parabola and continue until you find a minimum close to the actual minimum.

7.3 Using the derivative

Derivatives are great for figuring out in which sub-interval one should consider a bracket (i.e. in a, b, c, do you bracket (a, b) or (b, c)). Which helps finding the minimum faster, as $f' < 0$.

7.4 Multidimensional minimization

Without the derivative the simplest method[4] to minimize a well-behaved function is probably the *simplex algorithm*, however it will be slower to find the *local* minimum. A simplex is a structure in N-dimensional space connecting $N + 1$ points (i.e. a triangle connecting three points in 2D space). The idea is to move through space down the slope and adapting its shape as an amoeba, until minimum is found. Note: the simplex method will zoom in to a local minimum.

[4]Interesting read: `http://solar.physics.montana.edu/kankel/ph567/examples/Octave/minimization/amoeba/NotesPh567.pdf`

Nelder-Mead simplex method: The method[5] then performs a sequence of transformations of the working simplex, aimed at decreasing the function values at its vertices. At each step, the transformation is determined by computing one or more test points, together with their function values, and by comparison of these function values with those at the vertices. This process is terminated when the working simplex becomes sufficiently small in some sense, or when the function values are close enough in some sense (provided f is continuous). The Nelder-Mead algorithm typically requires only one or two function evaluations at each step, while many other direct search methods use n or even more function evaluations.

7.5 MCMC

Definition of MCMC: "a Monte Carlo (random) sampling from a distribution where each new sample is drawn with some reference to the last (Markov Chain)". MCMC is a stochastic (random) approach. One samples $x_1, \ldots, x_n$ from the search space and returns where $f(x)$ is minimal. For the sampling Markov Chains enter the picture, often using a specific algorithm such as Metropolis-Hastings. MH works as follows: choose an arbitrary point $\mathbf{x}_0$ as the first sample, and pick a prob. density $g(\mathbf{x}|\mathbf{y})$ that suggests a candidate for the next sampled value $\mathbf{x}$, given the previously sampled $\mathbf{y}$ (often we use a Gaussian centered at $\mathbf{y}$). Then in each subsequent iteration t we generate $\mathbf{x}$ given $g(\mathbf{x}|\mathbf{x}_t)$, and calculate the acceptance ratio $\alpha = f(\mathbf{x})/f(\mathbf{x}_t)$. If $\alpha < 1$ accept the value, otherwise try again using the previous value.

MCMC side notes:

- Often, MCMC (+ MH) is used to sample posterior distributions
- In that case the acceptance/rejection conditions are slightly altered
- We define $\alpha = \min\{1, f(\mathbf{x})/f(\mathbf{x}_t)\}$ and generate a number $u \in [0, 1]$ uniformly
- If $u < \alpha \rightarrow$ accept, otherwise reject!

[5]Explanation from `http://www.scholarpedia.org/article/Nelder-Mead_algorithm`