Lecture Content Summaries for MATH3201 (from week 4)

Travis Mitchell

Updated: 21 March, 2019

Chapter 1

1 Week 4

1.1 Recap of the first three weeks

Started by looking at *Taylor series* approximations for a function value $f(x)$ near a known point $f(a)$:

$$f(x) \approx f(a) + (x-a)f'(a) + \frac{1}{2}(x-a)^2 f''(a) + \frac{1}{3!}(x-a)^3 f^{(3)} + \cdots + \frac{1}{n!}(x-a)^n f^{(n)}(a) \tag{1.1}$$

This we refer to as a Taylor approximation up to order n. This is how our expansion of $f(x)$ about $f(a)$ looks for a uni-variable function, but what about multivariate systems? An example for a two dimensional system would mean formulating our expansion of $f(x,y)$ about a point (a,b) which looks like:

$$\begin{aligned} f(x,y) =& f(a,b) + (x-a)f_x(a,b) + (y-b)f_y(a,b) + \\ & \frac{1}{2!}\left[(x-a)^2 f_{xx}(a,b) + 2(x-a)(y-b)f_{xy}(a,b) + (y-b)^2 f_{yy}(a,b)\right] + \text{H.o.T} \end{aligned} \tag{1.2}$$

This we can then write in a more general setting for a system of N variables by using our gradient and Hessian operators:

$$f(\mathbf{x}) = f(\mathbf{a}) + (\mathbf{x}-\mathbf{a})^T \nabla f(\mathbf{a}) + \frac{1}{2}(\mathbf{x}-\mathbf{a})^T H(\mathbf{a})(\mathbf{x}-\mathbf{a}) + \text{H.o.T} \tag{1.3}$$

With these expressions, we now have the base level of knowledge required to start looking at approximating derivatives. Initially, we looked at three approximations for the first derivative of a function $f(x)$ as well as an approximation for the second order derivative. The expressions for these are given in Table 1.1.

Table 1.1: Expressions and the associated order of error for first and second order derivative approximations.

Approximation	Expression	Order Error
1st order forward difference	$f'(x) \approx \frac{f(x+h)-f(x)}{h}$	$O(h)$
1st order backward difference	$f'(x) \approx \frac{f(x)-f(x-h)}{h}$	$O(h)$
1st order centred difference	$f'(x) \approx \frac{f(x+h)-f(x-h)}{2h}$	$O(h^2)$
2nd order difference	$f''(x) \approx \frac{f(x+h)-2f(x)+f(x-h)}{h^2}$	$O(h^2)$

From here, we investigated possible techniques for finding higher order derivatives and the value of leading order error terms. In particular an example using the **Method of Undetermined Coefficients** was given in W2L1. Additionally, the natural extension for partial derivatives was given, e.g.

$$f_x(x,y) \approx \frac{f(x+h,y) - f(x-h,y)}{2h}. \tag{1.4}$$

The next topic introduced was *root finding* in the real domain, $\mathscr{R}^n$. Namely, this means we are wanting to solve the expression $f(x) = 0$, for which Newton's method provides a nice follow on from Taylor series expansions. In order to use Newton's method, we first make an initial guess, x_0, and then iterate using,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{1.5}$$

Here it is clear that the derivative of the function cannot be zero at x_n, and we also need to compute the derivative (note that this can be done with the approximations discussed if an analytical form cannot be found). When considering this method, be sure to understand its limitations and times when it can go unstable!

The method can be extended to solve algebraic systems as well by making use of the Jacobian, $\mathscr{J}(\mathbf{x})$ and the update rule,

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathscr{J}^{-1}(\mathbf{x}_n) \cdot F(\mathbf{x}_n), \tag{1.6}$$

where $F(\mathbf{x})$ is the vector of functions, $F(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_n(\mathbf{x}))^T$.

The issue with this method comes from the computational expense of computing the inverse of the Jacobian.

A further extension of Newton's method can be used to solve for maxima and minima of functions, namely we would be solving for $\nabla f = 0$, rather than $f(\mathbf{x}) = 0$. For this, the Newton update would become:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \left[\mathscr{H}(f(\mathbf{x}_n))\right]^{-1} \cdot f(\mathbf{x}_n), \tag{1.7}$$

where $\mathscr{H}$ is the Hessian matrix. To then classify the result, we use the eigenvalues of the Hessian matrix,

$$\text{if } \begin{cases} \lambda_i > 0 \quad \forall \quad i, \implies \text{minimum} \\ \lambda_i < 0 \quad \forall \quad i, \implies \text{maximum} \\ \lambda_i \neq 0 \quad \forall \quad i \text{ but signs are not consistant}, \implies \text{saddle} \\ \lambda_i = 0 \quad \forall \quad i, \implies \text{inconclusive.} \end{cases} \tag{1.8}$$

The method of steepest descent provides another means for finding local minima of a function $f(x)$. In this approach, we start from an initial guess, $\mathbf{x}_0$, and move in the direction of steepest descent (as the name may suggest!). The update rule for this method is given as:

$$\mathbf{x}_{n+1} = \mathbf{x} - \lambda \nabla f \tag{1.9}$$

Here we compute $\mathbf{x}_{n+1} = \mathbf{x}_{n+1}(\lambda)$ and then substitute this into $f^* = f(\mathbf{x}_{n+1}(\lambda))$. From here, we solve λ for,

$$\frac{d}{d\lambda} f^* = 0, \tag{1.10}$$

and use this to solve for $\mathbf{x}_{n+1}$.

1.2 With that out of the way, Week 4 content - *Integration*!

The first technique for numerical integration that we introduce is the *Trapezoidal rule*, this consists of dividing the domain up into Trapezoids and then summing the area of these,

$$\int_a^b f(x)dx \approx h\left[0.5(f_0 + f_N) + \sum_{i=1}^{N-1} f_i\right]. \tag{1.11}$$

In order to analyse the error, we can look at a single trapezoid and use a Taylor expansion then compare this to the exact integral. From this, we obtain an error of $O(h^3)$, but be a little careful here as this is the

associated error for a single trapezoid. Summing these over the entire domain gives the global error as $O(h^2)$.

To improve on the accuracy of the Trapezoidal rule, we can also use a local quadratic approximation for the function and this is called *Simpson's rule*. This means we need three points for the approximation, so locally becomes:

$$\int_{x_i}^{x_{i+2}} f(x)dx \approx \frac{h}{3}(f_i + 4f_{i+1} + f_{i+2}) + O(h^5). \tag{1.12}$$

Then again, summing these sub intervals over the full domain increases the order of error globally to, $O(h^4)$.

The question now comes, is there a smart way to choose the step size h? This is where *adaptive quadrature* comes into play. This technique allows us to have the computer determine whether finer resolution is necessary at certain points in our domain. For example, with the Trapezoidal rule, we can compare the integral for a single trapezoid with that of the same trapezoid computed with 2 half intervals. The error here can effectively be compared to a tolerance to determine if sufficient resolution has been used.

2 Week 5

2.1 Adaptive integration

If may be common that we have a function or data set that we need to integrate, but there could be an 'uninteresting' region as well as regions of rapid change - using a constant interval size for the full domain for such a circumstance can be computationally excessive. Additionally, it can be simpler and more convenient for a user to input in a tolerance they want to achieve with the integration rather than specifying an interval number.

Adaptive trapezoidal rule

Compare integration value for a region using a single trapezoid, $S_{[x_i,x_i i+1]}$, compared to two, $S_{[x_i,u_i]}+S_{[u_i,x_{i+1}]}$. From this we find that:

$$\text{error}(S_{[x_i,x_i i+1]} - S_{[x_i,u_i]} - S_{[u_i,x_{i+1}]}) \approx 3 \times \text{error}(S_{[x_i,u_i]} + S_{[u_i,x_{i+1}]}) \tag{1.13}$$

Therefore, we can take our criterion for convergence as:

$$|S_{[x_i,x_i i+1]} - S_{[x_i,u_i]} - S_{[u_i,x_{i+1}]}| < 3 \times tol \tag{1.14}$$

Note that to achieve a specified tolerance, we need to account for the expected error when we half an interval. Therefore, when the algorithm fails the above criterion and we half our interval the required tolerance must also be halved.

Adaptive Simpson's rule

Compare integration value for a region using a single trapezoid, $S_{[x_i,x_i i+1]}$, compared to two, $S_{[x_i,u_i]}+S_{[u_i,x_{i+1}]}$. From this we find that:

$$\text{error}(S_{[x_i,x_i i+1]} - S_{[x_i,u_i]} - S_{[u_i,x_{i+1}]}) \approx 15 \times \text{error}(S_{[x_i,u_i]} + S_{[u_i,x_{i+1}]}), \tag{1.15}$$

but as a conservative rule of thumb, 15 is replaced by 10.

2.2 Ordinal differential equations

Wanting to work out numerical ways to solve problems of the type:

$$\frac{dy}{dt} = f(t,y) \tag{1.16}$$

given the initial condition, $y(t_0) = y_0$.

A simple way we can approach this is to expand about our initial condition with a Taylor series:

$$y(t_0+h) = y(t_0) + hy'(t_0) + O(h^2) \tag{1.17}$$
$$\approx y(t_0) + hf(t_0, y(t_0)) \tag{1.18}$$
$$\vdots \tag{1.19}$$
$$\implies y_{n+1} = y_n + hf(t_n, y_n) \tag{1.20}$$

Therefore, given our initial condition and the derivative we are able to progress the function $y(t)$ through time. This update rule is known as **Euler's method**, and in practice is not used due to the high order of error which adds up cumulatively as we move further in time. One possible option is to Taylor expand to higher orders, but in practice is also not done - however, from this thinking leads the family of Runge-Kutta methods for solving ODEs.

Runge-Kutta methods

To obtain these, we often perform an expansion with Taylor series and then look at using the chain rule to breakdown higher order derivatives. Some common solvers that we will use in this class include:

- Runge-Kutta method of order 2 (also referred to as the midpoint rule)

$$y_{n+1} = y_n + hf\left(t_n + \frac{h}{2}, y_n + \frac{h}{2} f(t_n, y_n)\right) + O(h^3) \tag{1.21}$$

- Runge-Kutta method of order 4

$$\begin{aligned} y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\ k_1 &= f(t_n, y_n) \\ k_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2} k_1\right) \\ k_3 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2} k_2\right) \\ k_4 &= f\left(t_n + h, y_n + h k_3\right) \end{aligned} \tag{1.22}$$

The general method for deriving Runge-Kutta methods is given in Wk6 L1.

3 Week 6

3.1 Stiff problems - implicit methods

Here, we formulated the integral over a section of the ODE by taking the height of a rectangle created at $y(t_{n+1})$, rather than t_n which led to the forwards Euler method. By doing this, we came up with the implicit Euler method;

$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1}) \tag{1.23}$$

Taking this approach leads us to a system of equations to solve, for which something like Newton's method can be used to solve for the solution. This requires more computation than the forward Euler method, but has improved stability properties making it essential for certain problems.
So what makes a problem stiff?

- "If a numerical method with a finite region of absolute stability applied to a system with any initial condition is forced to use, in a certain interval of integration, a step length which is excessively small in relation to the smoothness of the exact solution in that interval, then the system is said to be stiff in that interval;"
- Stiffness occurs when stability requirements, rather than those of accuracy, constrain the step size;
- Stiffness occurs when some components of the solution decay much more rapidly than others.

3.2 Systems of ODEs

As most higher order ODEs can be written as a system of first order ODEs, we can use previously analysed methods to solve these. For example we can solve using Euler methods, but now we have vectors,

$$\mathbf{y}^{n+1} = \mathbf{y}^n + h\mathbf{f}(\mathbf{y}^n, t^n), \tag{1.24}$$

where $\mathbf{y} = \{y_1, y_2, ..., y_n\}$, and $\mathbf{f} = \{f_1, f_2, ..., f_n\}$.

This can also be applied with higher order methods, such as the RK2 and RK4, previously shown for 1 variable.

3.3 Boundary value problems for ODEs

Shooting method

Here, rather than being given initial values allowing us to predict forward in time, here we are now given a start and end point that our solution must comply with, in general we can write this as,

$$\frac{d^2y}{dx^2} = f\left(x, y(x), \frac{dy}{dx}(x)\right), \quad y(a) = y_a \text{ and } y(b) = y_b. \tag{1.25}$$

The methodology is to first rewrite the second order problem as a system of first order ODEs. We then guess, for example, the initial derivative and use this to 'shoot' at the final solution with an ODE solver. This process can be written into its own function, returning an error of our shoot compared to the boundary value, and hence a root finding method can be used to solve.

Chapter 2

1 Week 8

This section of the course looks at three topics:

1. Boundary Value Problems
   (a) For ODEs and for PDEs;
   (b) Using shooting method and finite difference methods.
2. Interpolation
   (a) For finding a function that fits a data set;
   (b) Using polynomials and splines.
3. Integration
   (a) Using Newton's method, adaptive quadrature, polynomials.

1.1 The Shooting Method Continued

The shooting method solves the BVP by finding the IVP that has the same solution, namely we construct the BVP as:

$$y" = f(x, y, y') \tag{2.1}$$
$$\text{with } y(a) = a, \quad y'(a) = s^*. \tag{2.2}$$

This IVP is solved and the resultant value, $y(b)$, is compared with the given boundary, y_b:

$$F(s) = [y(b) - y_b]. \tag{2.3}$$

We then seek the solution where, $F(s) = 0$. Thus, we have rewritten the problem as a root finding situation that will give us an initial velocity to shoot at our given boundary value.

1.2 Finite Difference Method (FDM)

Instead of rewriting the problem as an equation for which we need to find its root, with the FDM we try to write a system of equations that will allow us to solve the problem. To do this, we replace the continuous derivatives with their discrete approximations:

$$y' = \frac{y(x+\Delta x) - y(x-\Delta x)}{2\Delta x} \tag{2.4}$$
$$y'' = \frac{y(x+\Delta x) - 2y(x) + y(x-\Delta x)}{\Delta x^2} \tag{2.5}$$

These approximations are then substituted into our original PDE and applied at discrete points across the domain to be solved. This effectively gives us a closed set of simultaneous equations. Here, the boundaries y_0, y_N are given and we are left with $N-1$ equations and unknowns, $y_1, ..., y_{N-1}$.

From here, the problem becomes an issue of solving for $\mathbf{x}$ in a general system, $A\mathbf{x} = b$. In this, A is the coefficient matrix (typically diagonal), $\mathbf{x}$ is our set of unknown variables and b is a known vector.

Non-Linear BVP

If we have a non-linear system, we can no longer write the problem in the form, $A\mathbf{x} = b$. Here we again turn it into a bit of a root finding scenario:
e.g.

$$y" = 18y^2 \tag{2.6}$$

$$y_{i+1} - 2y_i + y_{i-1} = 18h^2 y_i^2 \tag{2.7}$$

Rewrite in the form, $F(\mathbf{y}) = 0$:

$$y_{i+1} - 2y_i - 18h^2 y_i^2 + y_{i-1} = 0 \tag{2.8}$$

We then look to use Newton's method to solve the system, $F_1, ...F_{N-1} = \mathbf{0}$. Remember Newton's method for a system of equations is given by:

$$\mathbf{y}^{j+1} = \mathbf{y}^j - J^{-1}(\mathbf{y}^j)F(\mathbf{y}^j) \tag{2.9}$$

2 Week 9

2.1 Partial Differential Equations

The Diffusion Equation

To start with, we will look at methods to solve the diffusion equation:

$$\partial_t u = D\partial_{xx} u \tag{2.10}$$

This can be used to govern the diffusion of heat in a bar or contaminant in a still room. To solve this problem, we need an initial state, $u(x,0)$, as well as boundary conditions, $u(a,t)$ and $u(b,t)$. To solve this, there are a few approaches, here we wills tart with the *Forward Difference Method* - i.e. we take a forward difference for the temporal derivative.

$$\text{Let } u(x_i, t_j) = u_{i,j} \tag{2.11}$$

$$\frac{u_{i,j+1} - u_{i,j}}{\Delta t} = D\left(\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}\right) \tag{2.12}$$

$$u_{i,j+1} = u_{i,j} + \sigma\left(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}\right), \qquad \sigma = \Delta t D/\Delta x^2 \tag{2.13}$$

Here we have explicitly written the next point in time directly from a previous time step. Explicit methods typically have stability constraints, for this particular case the constraint is:

$$\sigma \leq 0.5 \tag{2.14}$$

To remove the constraint relating Δt and Δx, we can look at implicit methods - namely, the *Backwards Difference Method*:

$$\text{Let } u(x_i, t_j) = u_{i,j} \tag{2.15}$$

$$\frac{u_{i,j} - u_{i,j-1}}{\Delta t} = D\left(\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}\right) \tag{2.16}$$

$$(1+2\sigma)u_{i,j} - \sigma(u_{i+1,j} + u_{i-1,j}) = u_{i,j-1} \tag{2.17}$$

This can be written in matrix form, $B\mathbf{u}^{(j)} = \mathbf{u}^{(j-1)}$, for which we can solve to progress intime by taking the inverse of the matrix, B. This turns out to be unconditionally stable, although calculating the inverse of a large matrix can potentially be costly.

The Wave Equation

This Wave equation is used to describe, for example, the propagation of a wave along a string:

$$\partial_{tt} u = c^2 \partial_{xx} u, \tag{2.18}$$

where $u = u(x,t)$, c is the speed of sound.

Therefore, we can replace our continuous derivatives with their discrete approximations:

$$u_{i,j+1} - 2u_{i,j} + u_{i,j-1} = \frac{c^2 \Delta t^2}{\Delta x^2}\left(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}\right) \tag{2.19}$$

For this case, the CFL number is given by,

$$\sigma = \frac{c\Delta t}{\Delta x} \leq 1. \tag{2.20}$$

3 Week 10

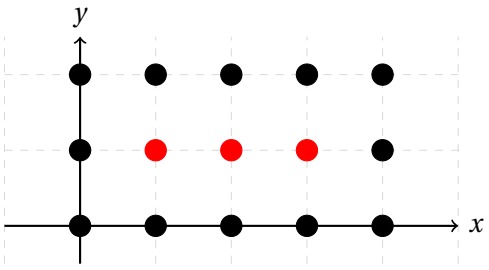
The Wave Equation conti.

Went through one method to solve the wave equation analytically, and came up with **D'Alembert's formula:**

$$u(x,t) = 0.5\left(h(x+ct) + h(x-ct)\right) + \frac{1}{2c}\int_{x-ct}^{x+ct} k(v)\,dv, \tag{2.21}$$

where $u(x,0) = h(x)$ and $\partial_t u(x,0) = k(x)$.

So how do we solve equation 2.19? As we now have a larger *stencil* than when we were looking at the diffusion/heat equation, we do need additional boundary conditions. In the lecture example the values surrounding the time/space domain were given, which left the interior as unknowns that we could write equations for - shown below with red unknown circles, black are known boundary values.



Laplace Equation

This can be used to describe the stead-state of (for example) the heat distribution on a surface:

$$\partial_{xx}u + \partial_{yy}u = 0 \tag{2.22}$$
$$\Delta u = 0 \tag{2.23}$$
$$\nabla^2 u = 0 \tag{2.24}$$

If the RHS of these equations is a function, $f(x,y)$ we get the **Poisson Equation**. This is an import equation to solve and comes up in a number of areas (e.g solving pressure fields in computational fluid dynamics). So how do we solve this using the *finite difference method*? Therefore, we wish to solve:

$$\nabla^2 u = f(x,y) \tag{2.25}$$

Subject to Dirichlet (fixed value of u) and Neumann (fixed derivative of u) boundary conditions. Here we can think of the Neumann B.C. as a flux into our system. For simplicity, let us take $\Delta x = \Delta y = h$, than the discrete equation will become:

$$u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j} - h^2 f_{i,j} = 0 \tag{2.26}$$

Therefore, we can do a similar domain deconstruction as with the wave equation and write algebraic expressions for each unknown grid point.

4 Week 11

Laplace / Poisson's Equation Continued... Iterative solution methods!

For this, we are going to look at three methods of solving our system of unknowns created when we write out the discrete form of the Laplace/Poisson equations for each node in our discrete domain:

1. Jacobi method;
2. Gauss-Seidel method;
3. Successive over-relaxation method.

Jacobi's Method

This is perhaps the simplest method for solving systems such as, $A\mathbf{x} = b$, and gives a good basis for understanding more complex iterative schemes. In the past this method has not been overly used for practical solutions, however, with the advent of parallel programming - this is being reconsidered. To apply this scheme, we rearrange Equation 2.26 to obtain,

$$u_{i,j} = \frac{1}{4}\left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - h^2 f_{i,j}\right). \tag{2.27}$$

From here, we make an initial guess, $u^0_{i,j}$ and use this equation to update successive values of $u_{i,j}$ until we converge,

$$u^{(n+1)}_{i,j} = \frac{1}{4}\left(u^{(n)}_{i+1,j} + u^{(n)}_{i-1,j} + u^{(n)}_{i,j+1} + u^{(n)}_{i,j-1} - h^2 f_{i,j}\right). \tag{2.28}$$

The good thing about this method, is that it always converges, however, the convergence can be very slow.

Gauss-Seidel Method

This method is a small variation on Jacobi's method in which, once we have calculated $u_{i,j}^{(n+1)}$, why not use it for the next point that we calculate? Namely, if we look at a point located at $(i,j) = (2,2)$, we have most likely already updated $u_{1,2}^{(n+1)}$ and $u_{2,1}^{(n+1)}$:

$$u_{2,2}^{(n+1)} = \frac{1}{4}\left(u_{3,2}^{(n)} + u_{1,2}^{(n+1)} + u_{2,3}^{(n)} + u_{2,1}^{(n+1)} - h^2 f_{2,2}\right). \tag{2.29}$$

Successive Over-Relaxation (SoR) Method

Here we look to use a weighted average of the new and old values in our solution such that the previous example would become,

$$u_{2,2}^{(n+1)} = (1-\omega)u_{2,2}^{(n)} + \frac{\omega}{4}\left(u_{3,2}^{(n)} + u_{1,2}^{(n+1)} + u_{2,3}^{(n)} + u_{2,1}^{(n+1)} - h^2 f_{2,2}\right). \tag{2.30}$$

For this, we have,

$$\begin{cases} \omega < 1, & \text{under-relaxation} \implies \text{slows down convergence} \\ \omega = 1, & \text{Gauss-Seidel} \\ \omega > 1, & \text{over-relaxation} \implies \text{speeds up convergence} \end{cases}$$

Aside: if we have flux boundary conditions, we can use the second order, three-part approximation,

$$f'(x) = \frac{-3f(x) + 4f(x+h) - f(x+2h)}{2h} + O(h^2) \tag{2.31}$$

4.1 Interpolation

The aim we have with interpolation for this section is to find a function that passes through a given data set. This allows us to determine values between data points as well as find lines of best fit.
Definition:

A function, $y = f(x)$, interpolates data points, (x_i, y_i), with $1 \le i \le n$ if $f(x_i) = y_i \ \forall i$.

Here, we will discuss two methods of interpolation,

1. Polynomials:
   This is useful as it gives us 'straight forward' mathematical properties and allows us to evaluate complicated functions with elementary operators.

   - Lagrangian Interpolation
     Given n data points $(x_1, y_1), \ldots, (x_n, y_n)$, we can define a polynomial that interpolates this,

$$P(x) = \sum_{i=1}^{n} y_i L_i(x), \quad \text{where} \tag{2.32}$$
$$L_i(x) = \frac{(x-x_1)\ldots(x-x_{i-1})(x-x_{i+1})\ldots(x-x_n)}{(x_i-x_1)\ldots(x_i-x_{i-1})(x_i-x_{i+1})\ldots(x_i-x_n)}$$

   - Newton's Divided Differences
     Let (x_i, y_i) where $i = 1, \ldots, n$ be n data points with distinct x_i. There exists one and only one polynomial, P, of degree at most $(n-1)$ such that $P(x_i) = y_i$. For this, we can define the first divided difference,

$$f[x_0, x_1] = \frac{f(x_1) - f(x_0)}{x_1 - x_0}, \tag{2.33}$$

and the second divided diffence as,

$$f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}. \tag{2.34}$$

From this, we can define the polynomial that fits the data points as,

$$f(x) = f(x_0) + f[x_0, x_1] + (x - x_0)(x - x_1) f[x_0, x_1, x_2] + \dots \tag{2.35}$$

$$+ (x - x_0)(x - x_1)\dots(x - x_k) f[x_0, x_1, \dots, x_k] \tag{2.36}$$

5 Week 12

- Chebyshev Theorem
  Given n data points on the interval $[a, b]$, the interpolation error is minimised if n data points are moved to Chebyshev's points (roots),

$$x_i = \frac{b-a}{2} \cos\left(\frac{(2i-1)\pi}{2n}\right) + \frac{b+a}{2}, \qquad i = 1, \dots, n. \tag{2.37}$$

  This way (moving data points) the numerator error is minimised a determinable upper bound,

$$(x - x_1)(x - x_2)\dots(x - x_n) \leq \frac{(b-a)^n}{2^{2n-1}}. \tag{2.38}$$

2. Splines
   In polynomial interpolation we used a single polynomial to interpolate between all data points. With splines, we now use up to $(n-1)$ polynomials to interpolate n points. Here, it is common practice to use cubic polynomials, which is sufficient to satisfy;

   - spline is continuous;
   - spline is smooth;
   - spline has some curvature.

   For a *natural spline*, we also have the condition that $s_1''(x_1) = s_{n-1}''(x_{n-1}) = 0$.

   From all of these conditions, we end up with $3(n-1)$ equations and $3(n-1)$ unknowns. Namely, we have $(n-1)$ points and three coefficients at each to calculate. Therefore, we take our approximation,

$$s_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3, \qquad a_i = y_i, \tag{2.39}$$

   and seek to determine b_i, c_i, d_i. This is done by taking,

$$\delta_i = x_{i+1} - x_i \tag{2.40}$$

$$\Delta_i = y_{i+1} - y_i, \tag{2.41}$$

   such that we can define b, d in terms of c as,

$$d_i = \frac{c_{i+1} - c_i}{3 * J_i}, \tag{2.42}$$

$$b_i = \frac{\Delta_i}{\delta_i} - \frac{\delta_i}{3}(2c_i + c_{i+1}). \tag{2.43}$$

   To solve for c_i, the equations from the above spline conditions allow us to write,

$$\begin{pmatrix} 1 & 0 & 0 & \dots & \dots \\ \delta_1 & 2(\delta_1+\delta_2) & \delta_2 & 0 & \dots \\ 0 & \delta_2 & 2(\delta_2+\delta_3) & \delta_3 & \dots \\ \vdots & \ddots & \ddots & \ddots & \ddots \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ \vdots \\ c_n \end{pmatrix} = \begin{pmatrix} 0 \\ 3(\Delta_2/\delta_2 - \Delta_1/\delta_1) \\ \vdots \\ 3(\Delta_{n-1}/\delta_{n-1} - \Delta_{n-2}/\delta_{n-2}) \\ 0 \end{pmatrix} \tag{2.44}$$

5.1 Extrapolation

Rather than interpolating within the range of data we have, we will now look at methods for extrapolating beyond our given range.

Richardson's Extrapolation Method

Here the idea is to use multiple step sizes to improve the approximation,

$$v_t \approx v_a(h) + kh^n \tag{2.45}$$
$$v_t \approx v_a(h/2) + k(h/2)^n \tag{2.46}$$
$$\implies v_t(1-2^n) \approx v_a(h) - 2^n v_a(h/2), \tag{2.47}$$

therefore, our extrapolated approximation becomes,

$$v_t \approx v_a(h/2) + \frac{v_a(h/2) - v_a(h)}{2^n - 1}. \tag{2.48}$$

5.2 Numerical Integration

1. Newton-Cotes formulas, estimate integration with equally spaced points
   (a) Trapezoidal rule
   (b) Simpson's rule
   (c) Romberg rule (extrapolation.)
2. Adaptive quadrature
   Numerical integration is based on evaluating integral by adapting size of intervals.