Lecture 1: Lagrange Interpolation

This lecture adapted from the numerical analysis textbook by Süli and Mayers, Ch. 6.

Notation: Π_n = real polynomials of degree $\leq n$

Setup: given data f_i at distinct x_i, $i = 0, 1, ..., n$, with $x_0 < x_1 < ... < x_n$, can we find a polynomial p_n such that $p_n(x_i) = f_i$? Such a polynomial is said to **interpolate** the data.

E.G.: constant $n = 0$ linear $n = 1$ quadratic $n = 2$

Theorem. $\exists p_n \in \Pi_n$ such that $p_n(x_i) = f_i$ for $i = 0, 1, ..., n$.

Proof: Consider, for $k = 0, 1, ..., n$, the "cardinal polynomial"

$$L_{n,k}(x) = \frac{(x - x_0)...(x - x_{k-1})(x - x_{k+1})...(x - x_n)}{(x_k - x_0)...(x_k - x_{k-1})(x_k - x_{k+1})...(x_k - x_n)} \in \Pi_n.$$

Then

$$L_{n,k}(x_i) = 0 \text{ for } i = 0, ..., k-1, k+1, ..., n \text{ and } L_{n,k}(x_k) = 1.$$

So now define

$$p_n(x) = \sum_{k=0}^{n} f_k L_{n,k}(x) \in \Pi_n$$

therefore

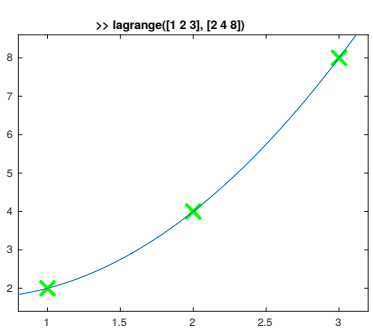
$$p_n(x_i) = \sum_{k=0}^{n} f_k L_{n,k}(x_i) = f_i \text{ for } i = 0, 1, ..., n.$$

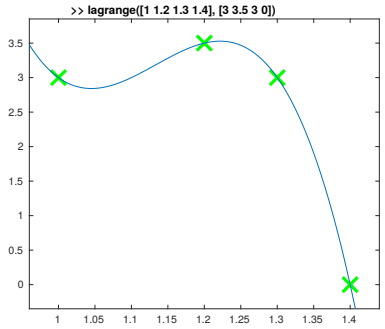
The polynomial is the **Lagrange interpolating polynomial**.

Theorem. The interpolating polynomial of degree $\leq n$ is unique.

Proof. Consider two interpolating polynomials $p_n, q_n \in \Pi_n$. Their difference $d_n = p_n - q_n \in \Pi_n$ satisfies $d_n(x_k) = 0$ for $k = 0, 1, ..., n$. i.e., d_n is a polynomial of degree at most n but has at least $n + 1$ distinct roots. Therefore, $d_n \equiv 0$, and $p_n = q_n$.

Matlab: [`lagrange.m`]





Data from an underlying smooth function: Suppose that $f(x)$ has at least $n+1$ smooth derivatives in the interval (x_0, x_n). Let $f_k = f(x_k)$ for $k = 0, 1, ..., n$, and let p_n be the Lagrange interpolating polynomial for the data (x_k, f_k), $k = 0, 1, ..., n$.

Error: how large can the error $f(x) - p_n(x)$ be on the interval $[x_0, x_n]$?

Theorem. For every $x \in [x_0, x_n]$ there exists $\xi = \xi(x) \in (x_0, x_n)$ such that

$$e(x) \equiv f(x) - p_n(x) = (x - x_0)(x - x_1)...(x - x_n)\frac{f^{(n+1)}(\xi)}{(n+1)!},$$

where $f^{(n+1)}$ is the $(n+1)$-st derivative of f.

Proof. Trivial for $x = x_k$, $k = 0, 1, ..., n$ as $e(x) = 0$. So suppose $x \neq x_k$. Let

$$\phi(t) \equiv e(t) - \frac{e(x)}{\pi(x)}\pi(t),$$

where

$$\pi(t) \equiv (t - x_0)(t - x_1)...(t - x_n) = t^{n+1} + \text{L.O.T.} \in \Pi_{n+1}.$$

Now note that ϕ vanishes at $n+2$ points x and x_k, $k = 0, 1, ..., n$. Therefore ϕ' vanishes at $n+1$ points $\xi_0, ..., \xi_n$ between these points. Therefore ϕ'' vanishes at n points between these new points, and so on until $\phi^{(n+1)}$ vanishes at an (unknown) point ξ in (x_0, x_n). But

$$\phi^{(n+1)}(t) = e^{(n+1)}(t) - \frac{e(x)}{\pi(x)}\pi^{(n+1)}(t) = f^{(n+1)}(t) - \frac{e(x)}{\pi(x)}(n+1)!$$

since $p_n^{(n+1)}(t) \equiv 0$ and because $\pi(t)$ is is a monic polynomial of degree $n+1$. The result then follows immediately from this identity since $\phi^{(n+1)}(\xi) = 0$.

Example: $f(x) = \log(1+x)$ on $[0, 1]$. Here, $|f^{(n+1)}(\xi)| = n!/(1+\xi)^{n+1} < n!$ on $(0, 1)$. So $|e(x)| < |\pi(x)|n!/(n+1)! \leq 1/(n+1)$ since $|x - x_k| \leq 1$ for each $x, x_k, k = 0, 1, ..., n$, in $[0, 1]$, therefore $|\pi(x)| \leq 1$. This is probably pessimistic for many x, e.g. for for $x = \frac{1}{2}, \pi(\frac{1}{2}) \leq 2^{-(n+1)}$ as $|\frac{1}{2} - x_k| \leq \frac{1}{2}$.

This shows the important fact that when using equally-spaced points, the error can be large at the end points, an effect known as the "Runge phenomena" (Carl Runge, 1901). Try demo `lec01 runge.m`.

Building Lagrange interpolating polynomials from lower degree ones.

Theorem. Let $Q_{i,j}$ be the Lagrange interpolating polynomial at x_k, $k = i, ..., j$. Then:

$$Q_{i,j}(x) = \frac{(x - x_i)Q_{i+1,j}(x) - (x - x_j)Q_{i,j-1}(x)}{x_j - x_i}.$$

Proof. Because of uniqueness, we simply wish to show the RHS interpolates the given data...

Comment: this can be used as the basis for constructing interpolating polynomials. In textbooks, often find topics such as the Newton form and divided differences.

Generalisation: Hermite interpolating polynomial matches function data and derivative data. Can also be constructed in terms of $L_{n,k}$.

Lecture 2: Newton–Cotes Quadrature

See Chapter 7 of Süli and Mayers.

Terminology: Quadrature ≡ numerical integration.

Setup: given $f(x_k)$ at $n+1$ equally spaced points $x_k = x_0 + k \cdot h$, $k = 0, 1, ..., n$, where $h = (x_n - x_0)/n$. Suppose that $p_n(x)$ interpolates this data.

Idea: does

$$\int_{x_0}^{x_n} f(x)dx \approx \int_{x_0}^{x_n} p_n(x)dx?$$

We investigate the error in such an approximation below, but note that

$$\int_{x_0}^{x_n} p_n(x)dx = \int_{x_0}^{x_n} \sum_{k=0}^{n} f(x_k) \cdot L_{n,k}(x)dx \tag{1}$$

$$= \sum_{k=0}^{n} f(x_k) \cdot \int_{x_0}^{x_n} L_{n,k}(x)dx \tag{2}$$

$$= \sum_{k=0}^{n} w_k f(x_k), \tag{3}$$

where the coefficients

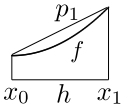
$$w_k = \int_{x_0}^{x_n} L_{n,k}(x)dx$$

$k = 0, 1, ..., n$, are independent of f . A formula

$$\int_a^b f(x)dx \approx \sum_{k=0}^{n} w_k f(x_k)$$

with $x_k \in [a, b]$ and w_k independent of f for $k = 0, 1, ..., n$ is called a quadrature formula; the coefficients w_k are known as weights. The specific form (1)–(3), based on equally spaced points, is called a Newton–Cotes formula of order n.

Examples: Trapezium Rule: $n = 1$ (also known as the trapezoid or trapezoidal rule):



$$\int_{x_0}^{x_1} f(x)dx \approx \frac{h}{2}[f(x_0) + f(x_1)]$$

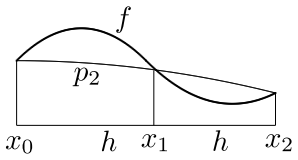
Proof

$$\int_{x_0}^{x_1} p_1(x)dx = f(x_0)\int_{x_0}^{x_1} L_{1,0}(x)dx + f(x_1)\int_{x_0}^{x_1} L_{1,1}(x)dx$$

$$= f(x_0)\int_{x_0}^{x_1} \frac{x - x_1}{x_0 - x_1}dx + f(x_1)\int_{x_0}^{x_1} \frac{x - x_0}{x_1 - x_0}dx$$

$$= f(x_0)\frac{(x_1 - x_0)}{2} + f(x_1)\frac{(x_1 - x_0)}{2}$$

Simpson's Rule: $n = 2$:



$$\int_{x_0}^{x_2} f(x)dx \approx \frac{h}{3}[f(x_0) + 4f(x_1) + f(x_2)]$$

Note: The trapezium rule is exact if $f \in \Pi_1$, since if $f \in \Pi_1$, therefore $p_1 = f$. Similarly, Simpson's Rule is exact if $f \in \Pi_2$, since if $f \in \Pi_2$, therefore $p_2 = f$. The highest degree of polynomial exactly integrated by a quadrature rule is called the (polynomial) degree of accuracy (or degree of exactness).

Error: we can use the error in interpolation directly to obtain

$$\int_{x_0}^{x_n} [f(x) - p_n(x)]dx = \int_{x_0}^{x_n} \frac{\pi(x)}{(n+1)!} f^{(n+1)}(\xi(n))dx$$

so that

$$\int_{x_0}^{x_n} [f(x) - p_n(x)]dx \leq \frac{1}{(n+1)!} \max_{\xi\in[x_0,x_n]} |f^{(n+1)}(\xi)| \int_{x_0}^{x_n} |\pi(x)|dx, \tag{4}$$

which, e.g., for the trapezium rule, $n = 1$, gives

$$\left| \int_{x_0}^{x_1} f(x)dx - \frac{(x_1 - x_0)}{2}[f(x_0) + f(x_1)] \right| \leq \frac{(x_1 - x_0)^3}{12} \max_{\xi\in[x_0,x_1]} |f''(\xi)|.$$

In fact, we can prove a tighter result:

Theorem. Error in Trapezoidal Rule:

$$\left| \int_{x_0}^{x_1} f(x)dx - \frac{(x_1 - x_0)}{2}[f(x_0) + f(x_1)] \right| = \frac{(x_1 - x_0)^3}{12} f''(\xi)$$

for some $\xi \in (x_0, x_1)$. (And note equality)

Proof. Omitted (uses Integral Mean-Value Theorem).

For $n > 1$, (4) gives pessimistic bounds. But one can prove better results, e.g., using Taylor Series.

Theorem. Error in Simpson's Rule: if f''' is continuous on (x_0, x_2), then

$$\left| \int_{x_0}^{x_2} f(x)dx - \frac{x_2 - x_0}{6}[f(x_0) + 4f(x_1) + f(x_2)] \right| = \frac{(x_2 - x_0)^5}{2880} f'''(\xi)$$

for some $\xi \in (x_0, x_2)$.

Proof. See, e.g., Süli and Mayers, Thm. 7.2.

Note: Simpson's Rule is exact if $f \in \Pi_3$ since then $f''' \equiv 0$. (c.f. earlier statement viz. $f \in \Pi_2$).

Composite Quadrature (optional material)

Motivation: we've seen oscillations in polynomial interpolation—the Runge phenomenon—for high-degree polynomials on equispaced grids.

Idea: split a required integration interval $[a,b]=[x_0,x_n]$ into n equal intervals $[x_{i-1},x_i]$ for $i=1,\ldots,n$. Then use a **composite rule**:

$$\int_a^b f(x)\,\mathrm{d}x=\int_{x_0}^{x_n} f(x)\,\mathrm{d}x=\sum_{i=1}^{n}\int_{x_{i-1}}^{x_i} f(x)\,\mathrm{d}x$$

in which each $\int_{x_{i-1}}^{x_i} f(x)\,\mathrm{d}x$ is approximated by quadrature.

Thus rather than increasing the degree of the polynomials to attain high accuracy, instead increase the number of intervals.

Composite Trapezium Rule:

$$\begin{aligned}\int_{x_0}^{x_n} f(x)\,\mathrm{d}x &= \sum_{i=1}^{n}\left[\frac{h}{2}[f(x_{i-1})+f(x_i)]-\frac{h^3}{12}f''(\xi_i)\right]\\ &= \frac{h}{2}[f(x_0)+2f(x_1)+2f(x_2)+\cdots+2f(x_{n-1})+f(x_n)]+e_h^{\mathrm{T}}\end{aligned}$$

where $\xi_i\in(x_{i-1},x_i)$ and $h=x_i-x_{i-1}=(x_n-x_0)/n=(b-a)/n$, and the error e_h^{T} is given by

$$e_h^{\mathrm{T}}=-\frac{h^3}{12}\sum_{i=1}^{n}f''(\xi_i)=-\frac{nh^3}{12}f''(\xi)=-(b-a)\frac{h^2}{12}f''(\xi)$$

for some $\xi\in(a,b)$, using the Intermediate-Value Theorem n times. Note that if we halve the stepsize h by introducing a new point halfway between each current pair (x_{i-1},x_i), the factor h^2 in the error will decrease by four.

Alternatively, divide $[a,b]$ into $2n+1$ intervals: $[a,b]=[x_0,x_{2n}]$. Then:

Composite Simpson's Rule:

$$\begin{aligned}\int_{x_0}^{x_{2n}} f(x)\,\mathrm{d}x &= \sum_{i=1}^{n}\left[\frac{h}{3}[f(x_{2i-2})+4f(x_{2i-1})+f(x_{2i})]-\frac{(2h)^5}{2880}f''''(\xi_i)\right]\\ &= \frac{h}{3}[f(x_0)+4f(x_1)+2f(x_2)+4f(x_3)+2f(x_4)+\cdots\\ &\qquad +2f(x_{2n-2})+4f(x_{2n-1})+f(x_{2n})]+e_h^{\mathrm{S}}\end{aligned}$$

where $\xi_i\in(x_{2i-2},x_{2i})$ and $h=x_i-x_{i-1}=(x_{2n}-x_0)/2n=(b-a)/2n$, and the error e_h^{S} is given by

$$e_h^{\mathrm{S}}=-\frac{(2h)^5}{2880}\sum_{i=1}^{n}f''''(\xi_i)=-\frac{n(2h)^5}{2880}f''''(\xi)=-(b-a)\frac{h^4}{180}f''''(\xi)$$

for some $\xi\in(a,b)$. Note that if we halve the stepsize h by introducing a new point half way between each current pair (x_{i-1},x_i), the factor h^4 in the error will decrease by sixteen.

Adaptive (or automatic) procedure: if S_h is the value given by composite Simpson's rule with a stepsize h, then

$$S_h - S_{\frac{1}{2}h} \approx -\frac{15}{16} e_h^{\mathrm{S}}.$$

This suggests that if we wish to compute $\displaystyle\int_a^b f(x)\,\mathrm{d}x$ with an absolute error ε, we should compute the sequence $S_h, S_{\frac{1}{2}h}, S_{\frac{1}{4}h}, \ldots$ and stop when the difference, in absolute value, between two consecutive values is smaller than $\frac{16}{15}\varepsilon$. That will ensure that (approximately) $|e_h^{\mathrm{S}}| \leq \varepsilon$.

Sometimes much better accuracy may be obtained: for example, as might happen when computing Fourier coefficients, if f is periodic with period $b-a$ so that $f(a+x) = f(b+x)$ for all x.

Matlab:

```
>> help adaptive_simpson
 ADAPTIVE_SIMPSON  Adaptive (or automatic) Simpson's rule.
    S = ADAPTIVE_SIMPSON(F,A,B,NMAX,TOL) computes an approximation
    to the integral of F on the interval [A,B]. It will take a
    maximum of NMAX steps and will attempt to determine the
    integral to a tolerance of TOL.

    The function uses an adaptive Simpson's rule, as described
    in lectures.
>> f = @(x) sin(x);
>> adaptive_simpson(f, 0, pi, 100, 1e-7);
Step 1 integral is 2.0943951024, with error estimate 2.0944.
Step 2 integral is 2.0045597550, with error estimate 0.089835.
Step 3 integral is 2.0002691699, with error estimate 0.0042906.
Step 4 integral is 2.0000165910, with error estimate 0.00025258.
Step 5 integral is 2.0000010334, with error estimate 1.5558e-05.
Step 6 integral is 2.0000000645, with error estimate 9.6884e-07.
Successful termination at iteration 7:
The integral is 2.0000000040, with error estimate 6.0498e-08.

>> g = @(x) sin(sin(x));
>> fplot(g, [0 pi])
>> adaptive_simpson(g, 0, pi, 100, 1e-7);
Step 1 integral is 1.7623727094, with error estimate 1.7624.
Step 2 integral is 1.8011896009, with error estimate 0.038817.
Step 3 integral is 1.7870879453, with error estimate 0.014102.
Step 4 integral is 1.7865214631, with error estimate 0.00056648.
Step 5 integral is 1.7864895607, with error estimate 3.1902e-05.
Step 6 integral is 1.7864876112, with error estimate 1.9495e-06.
Step 7 integral is 1.7864874900, with error estimate 1.2118e-07.
Successful termination at iteration 8:
The integral is 1.7864874825, with error estimate 7.5634e-09.
```

Lecture 3: Differentiation

So far we're looked at interpolation and quadrature. In both cases, we're given data f_i at points x_i and we want to compute/estimate/approximate:

1. the function values (interpolation);
2. the definite integral (quadrature).

What about differentiation? Suppose we want to estimate the derivative of a function at a point (often, but not always, one of the x_i's).

Fit interpolating polynomial and differentiate it

This approach is conceptually simple and illustrative, although traditionally not used very often in practice (however, search for "Barycentric Lagrange", or "Chebfun")

Example Suppose I want to estimate the second derivative of a function $f(x)$ at a point x_j from samples f_i and nodes x_i. How many data points do I need? Constant or linear interpolant won't work (why not?) Simplest is quadratic.

[Worked example, perhaps in a CAS...]

Assuming the data is equispaced, note this reduces to the gives the common "1 -2 1" rule:

$$f''(x_j) = \frac{1}{h^2} f_{j-1} - \frac{2}{h^2} f_j + \frac{1}{h^2} f_{j+1}.$$

Error analysis What can we learn from the polynomial interpolant error formula?

Not much!

In some sense, predicts $O(h)$ which is correct when the data is not equispaced. Equispaced, it should be $O(h^2)$, which we'll see this more precisely by the next method.

Method of Undetermined Coefficients

Instead of working with interpolants, the most commonly used alternative approach is the method of undetermined coefficients.

Reference: [Chapter 1 of LeVeque 2007 Textbook].

Based on Taylor series. Taylor expand $f(x+h)$ and $f(x-h)$ (and others) in small parameter h about x.

Examples Various derivatives from three data points. Consider a smooth function $u(x)$ then

$$u(x+h) = u(x) + hu'(x) + \frac{h^2}{2}u''(x) + \frac{h^3}{6}u'''(x) + \frac{h^4}{24}u''''(x) + \dots$$

$$u(x-h) = u(x) - hu'(x) + \frac{h^2}{2}u''(x) - \frac{h^3}{6}u'''(x) + \frac{h^4}{24}u''''(x) - \dots$$

From this can show how we can compute three difference *schemes* (approximations) to $u'(x)$:

Forward difference:

$$u'(x) = \frac{u(x+h) - u(x)}{h} + O(h)$$

Backward difference:

$$u'(x) = \frac{u(x) - u(x-h)}{h} + O(h)$$

Centered difference:

$$u'(x) = \frac{u(x+h) - u(x-h)}{2h} + O(h^2)$$

Error Analysis Note: this gives the $O(h^2)$ error term.

For practical algorithms: [Fornberg, Calculation of weights in finite difference formulas, SIAM Rev. 1998].

Lecture 4: Timestepping and ODEs

Last Day Review

Recall from last day, we can use Taylor expansions of a smooth function $u(x)$:

$$u(x+h) = u(x) + hu'(x) + \frac{h^2}{2}u''(x) + \frac{h^3}{6}u'''(x) + \frac{h^4}{24}u''''(x) + \dots$$

$$u(x-h) = u(x) - hu'(x) + \frac{h^2}{2}u''(x) - \frac{h^3}{6}u'''(x) + \frac{h^4}{24}u''''(x) - \dots$$

Using these and the *method of undetermined coefficients*, we can derive approximations to derivatives:

Forward difference:

$$u'(x) = \frac{u(x+h) - u(x)}{h} + O(h)$$

Backward difference:

$$u'(x) = \frac{u(x) - u(x-h)}{h} + O(h)$$

Centered difference:

$$u'(x) = \frac{u(x+h) - u(x-h)}{2h} + O(h^2)$$

Ordinary differential equations: ODE IVPs

ODE: ordinary differential eqn (i.e., just 1 independent variable)

IVP: initial-value problem

First-order IVP in standard form:

$$u' = f(t,u), \quad t > 0$$
$$u(0) = \eta \quad \text{(initial data)}$$

We're looking for the solution, a function $u(t)$ for $t > 0$.

Looks restrictive but this form is quite general:

1. It could describe a system of N ODEs.
   - $u(t)$ is an N-vector, so is η.
2. Equivalence to higher-order system.
   - Example: harmonic oscillator

     $$w'' = -w', w(0) = 1, w'(0) = 0$$

     Define $u_1 = w$, $u_2 = w'$, then we have

$$u_1' = u_2,$$
$$u_2' = -u_1.$$

with initial conditions $u_1(0) = 1$, $u_2(0) = 0$.

In this example ODE the variable t does not appear explicitly ($f(t, u)$ is just $f(u)$). Such an ODE is said to be **autonomous**.

Another example: the sun and moon and eight planets. Each has three coordinates of position and three of velocity. All together, that's an autonomous ODE IVP of dimension $N = 60$.

Need for numerical methods

Above simple examples are linear and can be solved analytically. Most **nonlinear ODEs**, however, cannot; we need *numerics*.

van der Pol equation, a nonlinear example

$$w'' + C(w^2 - 1)w' + w = 0,$$

for fixed $C > 0$. For $|w| > 1$, a damped oscillator; for $|w| < 1$, negatively damped. Big solutions decay; small solutions grow. As $t \to \infty$ we have convergence to a **limit cycle** of size $O(1)$.

Equivalent first-order system:

$$u_1' = u_2,$$
$$u_2' = -u_1 - C[(u_1)^2 - 1]u_2.$$

[m01_vanderpol.m]

Time discretization

A grid in time: $t_n = nk, k > 0$ a fixed **time step**. Often "Δt" or "τ" used instead.

```
 |-------|-------|-------|-------|-------|-----
t0=0    t1=k    t2=2k   ...
```

(often in practice k not fixed, and is changed automatically).

The exact solution (usually unknown) is $u(t)$ defined for all continuous values of t. We will find a numerical solution at these discrete points $v_n \approx u(t_n)$.

Forward Euler method

The simplest method is the forward Euler method:

$v^0 = \eta$ (initial condition)

$v^1 = v^0 + kf(v^0)$

$v^2 = v^1 + kf(v^1)$

... etc.

We are *time-stepping* with the formula

$$v^{n+1} = v^n + kf(t_n, v^n)$$

And note this is the "forward difference" we saw earlier. Matlab codes: [m02_euler.m], [m03_euler_graph.m].

We often want methods which are more accurate, or more stable, or have other features.

Methods with smaller errors

Under some assumptions (smooth enough solution) the forward Euler method has a (global) error which behaves like Ck (notation: $error = O(k)$).

Roughly speaking if we want 6 digits of accuracy, we probably need about one million steps. Often we want more accuracy for less work. Money for nuthin'!

Runge–Kutta and linear multistep methods

Compared to forward Euler, we often want methods which are more accurate, or more stable, or have other features. We'll do some analysis later, but for now let's list some other methods.

There are two main classes of methods: Runge–Kutta ("1-step"); Adams–[others] ("multistep")

Adams-Bashforth multistep formulas Notation: define $f^n = f(t_n, v^n)$.

$v^{n+1} = v^n + kf^n$: *forward Euler* again, accuracy $O(k)$

$v^{n+1} = v^n + \frac{k}{2}(3f^n - f^{n-1})$: accuracy $O(k^2)$

$v^{n+1} = v^n + \frac{k}{12}(23f^n - 16f^{n-1} + 5f^{n-2})$: accuracy $O(k^3)$

$v^{n+1} = v^n + \frac{k}{24}(55f^n - 59f^{n-1} + 37f^{n-2} - 9f^{n-3})$: accuracy $O(k^4)$

...

(an infinite sequence of such formulas)

Adams–Bashforth uses previous values, not just the current one. [modify our code].

Tricky to start up because extra values are needed.

Runge–Kutta methods Use temporary intermediate values (called *stage values*).

1. Forward Euler (again) $O(k)$
2. "Modified Euler" $O(k^2)$
   $a = kf(t_n, v^n)$,
   $b = kf(t_n + k/2, v^n + a/2)$,
   $v^{n+1} = v^n + b$.
3. "*The* Fourth-order Runge-Kutta Method" $O(k^4)$
   $a = kf(t_n, v^n)$,
   $b = kf(t_n + k/2, v^n + a/2)$,
   $c = kf(t_n + k/2, v^n + b/2)$,
   $d = kf(t_n + k, v^n + c)$,
   $v^{n+1} = v^n + \frac{1}{6}(a + 2b + 2c + d)$.
   If you could take just one formula to a desert island, this is it.

Higher-order RK formulas get very complicated. The coefficients of Runge-Kutta methods can be stored in a "Butcher Tableau". Matlab's |ode45| uses a method of Dormand and Prince 1980 (see pg178 of HairerWannerNorsett). Jim Verner also constructed high-order methods (and still does). See "DVERK User Guide" from 1976.

IVP codes in higher-level software

ODE codes are among the must successful algorithms (and software) in all of scientific computing. The key is that they are **adaptive** – varying step size, and sometimes order, automatically.

Matlab has many codes, but in my experience, most people use ode45 and ode15s. These differ in numerics but are basically the same in syntax. For info see

L. F. Shampine and M. W. Reichel, The Matlab ODE suite, SIAM J. Sci. Comput. 18 (1997), 1-22.

Analysis

We would like to understand how the error of these various methods behaves.

Some concepts we will need:

- consistency and local truncation error
- stability
- convergence

The local truncation error

Substitute the true solution of the differential equation into the discrete problem. It will not satisfy it exactly: the discrepancy is the called the *local truncation error (LTE).*

symbol: τ

Use Taylor expansions (very similar to the derivation of forward difference, etc). Exact solution unknown, assumed smooth.

Express LTE in "big Oh" notation in k.

Consistency

LTE to zero as $k \to 0$.

example: 2nd-order Adams-Bashforth

$$v^{n+1} = v^n + \frac{k}{2}(3f^n - f^{n-1})$$

To determine the LTE, it is important to write this in a form similar to the DE: $u' = f(t, u)$

$$\frac{v^{n+1} - v^n}{k} = \frac{1}{2}(3f^n - f^{n-1})$$

Now expand each with Taylor: $u(t_n + k)$ and $f(t_n - k)$ (remember $u' = f$):

$$u(t + k) = u + ku' + (1/2)k^2u'' + (1/6)k^3u''' + \dots$$

$$f(t - k) = u' - ku'' + (1/2)k^2u''' - \dots$$

$$\tau = LHS - RHS...$$

Substitute in, find the amount by which the exact solution $u(t_{n+1})$ fails to satify the discrete equation:

$$\tau = \frac{u(t_n + k) - u(t_n)}{k} \left(\frac{1}{2}(3f^n - f^{n-1}) \right) = \dots = -(5/12)k^2u''' + \dots$$

So the local truncation error is $\tau = O(k^2)$. And as this goes to zero as $k \to 0$, we can say this method is *consistent*.

Runge–Kutta

Doing this sort of Taylor analysis for RK leads to *order conditions*:

> "These computations, which are not reproduced in Kutta's 1905 paper (they are, however, in Heun 1900), are very tedious. And they grow enormously with higher-orders. We shall see... using an appropriate notation, they can become very elegant." [Hairer, Wanner, Norsett]

Aside: LTE and consistency for ODEs (optional)

There are two basic contradictory definitions of LTE in the literature (oops). Depends on where you apply the Taylor expansions. E.g., for forward Euler:

1. As we did above: apply the Taylor analysis to $\frac{u^{n+1} - u^n}{k} = f(u^n)$. This gives LTE of $O(k)$.
2. Apply the Taylor analysis to $u^{n+1} = u^n + kf(u^n)$. This is sometimes called the *one-step error*: under this, LTE is $O(k^2)$. But we apply $O(1/k)$ steps for an estimated/expected global error of $O(k)$.

Stable and unstable methods

Having a small LTE isn't sufficient to tell us that the sequence actually converges! Recall that the method will be used repeatedly: an unstable method is one that where each error (from the LTE, rounding error or whatever) builds up (e.g., over time).

A stable method is one where small errors are "damped out".

Here's an example of an unstable formula with order 3, try it and see how the computed solutions blow up as $k \to 0$.

$$v^{n+1} = -4v^n + 5v^{n-1} + k(4f^n + 2f^{n-1})$$

[m05_unstab.m: an example of a multistep method that is consistent... but unstable.]

[m05_unstab.m: also an example of a non-self-starting method: use forward Euler or exact solution, either way still unstable.]

Convergence

Convergence says the actual overall global error (say at $t = T_f$ some final time) decreases as $k \to 0$. This is the property we want.

Intuitively, global convergence requires both consistency and stability. And in many cases this is enough...

Fundamental Theorem

Fundamental theorem of finite difference methods (or "of numerical analysis"?)

Consistency + Stability <=> Convergence

We will see this again-and-again. The notion of stability might change, depending on the situation (generally stability harder than consistency).

Multistep methods For multistep methods, this is the **Dahlquist Equivalence Theorem**.

Absolute Stability for Runge–Kutta methods

Dahlquist test problem: $u' = \lambda u$, $u(0) = 1$.

Here λ is a complex number. Exact solution is $u(t) = \exp(\lambda t)$. Real part of λ negative gives decay in the exact solution.

We apply the numerical method to this test problem and:

If the solution grows without bound, we say **unstable**, otherwise, we say the numerical method is **stable**.

Example: forward Euler

$$v^{n+1} = v^n + k\lambda v^n,$$

$$v^{n+1} = (1 + k\lambda)v^n.$$

But we can apply this all the way back to t_0:

$$v^{n+1} = (1 + k\lambda)^{n+1} v^0 = (1 + k\lambda)^{n+1}.$$

Thus for forward Euler, the soln will blow-up as $n \to \infty$ unless $|1 + k\lambda| \leq 1$. If λ is real, this means

$$-2 \leq k\lambda \leq 0.$$

Or in the complex plane, we have a **absolute stability region** for forward Euler $z = k\lambda$ consisting of a unit disc centered at $z = -1$.

Absolute stablity
For a given λ, and a given numerical method, we need to choose k such that $k\lambda$ lives inside the stability region.

Different RK and MS methods have different absolute stability regions.

Why is this a reasonable test problem?

This analysis is not much use if it only applies to this one problem...

Works well if (in theory) we can linearize our problem, and apply an eigenvalue decomposition. Each component of the diagonalized linear system will then behave like the Dahlquist test problem.

Stiffness

The classical stability/consistency/convergence theory for ODEs was established by Dahlquist in 1956. Just a few years later it began to be widely appreciated that something was missing from this theory. Key paper: [Dahlquist, 1963]

(Chemists Curtiss & Hirschfelder [1952], used the term "stiff", which may actually have originated with the statistician John Tukey (who also invented "FFT" and "bit").

Definitions of stiffness

- A stiff ODE is one with widely varying time scales.
- More precisely, an ODE with solution of interest $u(t)$ is stiff when there are time scales present in the equation that are much shorter than that of $u(t)$ itself.

Neither is an ideal definition.

- A quote from HW: "Stiff equations are problems for which explicit methods don't work".

- My favourite: a stiff *problem* is one were implicit methods work better. (I learned this from Raymond Spiteri but probably its due to Gear.) C.W. Gear, 1982:

  > Although it is common to talk about "stiff differential equations," an equation *per se* is not stiff, a particular initial value problem for that equation may be stiff, in some regions, but the size of these regions depend on the initial values *and* the error tolerance.

Why it matters: regardless of stiffness, you'll get convergence as $k \to 0$ (Dahlquist's original theory).

But the result might be rubbish except when k is very small, because of modes $k\lambda$ that aren't in the stability region.

Example

ODE $u' = -\sin(t)$ with IC $u(0) = 1$ has solution $u(t) = \cos(t)$.

Change ODE to

$$u' = -100(u(t) - \cos(t)) - \sin(t),$$

then this *still* has solution $u(t) = \cos(t)$.

But the numerics are much different:

[m07_stiff_fe_be.m: a convergence study showing forward Euler convergences for very small k but is unstable for larger k.]

We can also linearize around the soln: let $u(t) = \cos(t) + w(t)$ and we get an ODE for $w(t)$ of $w' = -100w$ which does indeed have a very different *time scale* than $\cos(t)$.

Introduction to implicit methods

(Part of) the cure for stiffness? Use implicit methods. E.g., backward Euler method:

$$v^{n+1} = v^n + kf^{n+1} = v^n + kf(t_{n+1}, v^{n+1}).$$

Note v^{n+1}, *the thing we want to solve for*, is inside the function f –> implicit.

[m07_stiff_fe_be.m: implement backward Euler and note no instability]

A-stability

Recall the exact solution of $u' = \lambda u$ is stable (the solution itself, nothing about numerical methods) if $Re(\lambda) < 0$. Seems desirable to have numerical methods that are also stable in this way.

Want their linear stability regions to include the left-hand of the complex plane.

Defn [Dahlquist 1963]: a method, whose stability region includes the left-hand complex plane is called **A-stable**.

> We also expect these to work well for stiff problems b/c $k\lambda$ (for any $Re(\lambda) < 0$) will be in the stability region.

Dahlquist in 1979:

> I didn't like all these "strong", "perfect", "absolute", "generalized", "super", "hyper", "complete" and so on in mathematical definitions, I wanted something neutral; and having been impressed by David Young's "Property A", I choose the term "A-stable".

Example, backward Euler

$$v^{n+1} = v^n + kf(t_n + k, v^{n+1}) = v^n + kf^{n+1}$$

Using Dahlquist test problem, can show it is stable in the exterior of a unit disc centered at 1 in the *right*-half complex plane.

Hairer & Wanner: "The backward Euler method is a *very* stable method."

(There are also implicit multistep and implicit Runge-Kutta.)

Solving the system

Implicit methods have both v^{n+1} and f^{n+1}: we need to solve a linear or nonlinear system to advance the step. Linear algebra if linear, something like Newton's method if nonlinear. We do not discuss this issue further.

Lecture 5: Finite differences and PDEs

Partial Differential Equations (PDEs)

PDE: partial differential equation: ≥ 2 independent variables.

History:

1600s: calculus.

1700s, 1800s: PDEs all over physics, especially linear theory.

1900s: the nonlinear explosion and spread into physiology, biology, electrical engr., finance, and just about everywhere else.

The PDE Coffee Table Book

Unfortunately this doesn't physically exist (yet!). Fortunately, we can read many of the pages online: people.maths.ox.ac.uk/trefethen/pdectb.html

Notation:

- *Partial derivatives* $u_t = \frac{\partial u}{\partial t}$, $u_{xx} = \frac{\partial^2 u}{\partial x^2}$,
- *Gradient* (a vector) $\operatorname{grad} u = \nabla u = (u_x, u_y, u_z)$,
- *Laplacian* (a scalar) $\operatorname{lap} u = \nabla^2 u = \triangle u = \Delta u = u_{xx} + u_{yy} + u_{zz}$,
- *Divergence* (vector input, scalar output)

$\operatorname{div}(u, v, w)^T = \nabla \cdot (u, v, w)^T = u_x + v_y + w_z$.

(and combinations: e.g., $\Delta u = \nabla \cdot \nabla u$).

Examples

Laplace eq: $\Delta u = 0$ (elliptic)

Poisson eq: $\Delta u = f(x, y, z)$ (elliptic)

Heat or diffusion eq: $u_t = \Delta u$ (parabolic)

Wave eq: $u_{tt} = \Delta u$ (hyperbolic)

Burgers eq: $u_t = (u^2)_x + \epsilon u_{xx}$

KdV eq: $u_t = (u^2)_x + u_{xxx}$

Finite differences in space and time

The simplest approach to numerical soln of PDE is finite difference discretization in both space and time (if it's time-dependent).

Consider the heat or diffusion equation in 1D:

$$u_t = u_{xx}, \quad -1 < x < 1,$$

with *initial conditions*: $u(x,0) = u_0(x)$,

and *boundary conditions*: $u(-1,t) = u(1,t) = 0$.

Set up a regular grid with k = time step , h = spatial step

$$v_j^n \approx u(x,t).$$

A simple finite difference formula:

$$\frac{v_j^{n+1} - v_j^n}{k} = \frac{v_{j+1}^n - 2v_j^n + v_{j-1}^n}{h^2}.$$

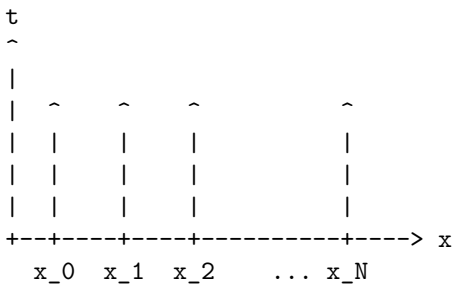
The Method-of-lines Discretize in space only, to get a system of ODEs. This reduces the problem to one we've already looked at: numerical solution of ODEs.

For the heat/diffusion equation, we get:

$$\frac{d}{dt}v_j = \frac{v_{j-1} - 2v_j + v_{j+1}}{h^2},$$

where $v_j(t)$ is a continuous function in time and $v_j(t) \approx u(x_j,t)$.

We solve the (coupled) ODEs along lines in time.



Consider v^n as an N-vector. We can get from v^n to v^{n+1} by using a matrix to approximate u_{xx}.

Linear algebra: MOL approach

$$\frac{d}{dt}\begin{pmatrix} v(t)_1 \\ \cdot \\ \cdot \\ \cdot \\ v(t)_N \end{pmatrix} = \frac{1}{h^2}\begin{pmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{pmatrix}\begin{pmatrix} v(t)_1 \\ \cdot \\ \cdot \\ \cdot \\ v(t)_N \end{pmatrix}$$

Or using $v(t)$ as N-vector:

$$\frac{d}{dt}v(t) = \frac{1}{h^2}\begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{bmatrix} v(t)$$

Let's call this matrix L (incl. $\frac{1}{h^2}$ factor) so that $\frac{d}{dt}v(t) = Lv(t)$. If we then discretize time with forward Euler we get:

$$v^{n+1} = v^n + k * \frac{1}{h^2}\begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{bmatrix} \begin{bmatrix} v^n_1 \\ v^n_2 \\ \vdots \\ v^n_{N-1} \\ v^n_N \end{bmatrix}$$

where v^{n+1} represents the discrete vector of solution at time t_{n+1}:

$$v^{n+1} \approx v(t_{n+1}).$$

We can also write the above as $v^{n+1} = Av^n$ with $A := I + kL$.

Can also convince ourselves that the above fully discrete matrix system is the same as the earlier fully discrete equations:

$$\frac{v_j^{n+1} - v_j^n}{k} = \frac{v_{j+1}^n - 2v_j^n + v_{j-1}^n}{h^2}.$$

(Note however that not all finite difference schemes are methods-of-lines).

Implementation of these matrices and schemes

[m10_heat_mol.m]

Discuss constructing the matrix L using sparse matrices. Sparse matrices save storage space here.

Matlab: discuss $k * L * u$ versus the better $k * (L * u)$.

You can also implement using for-loops. [m11_heat_loops.m] This can be much slower in Matlab (and other high-level languages), although "JIT-compilers" often speed it up.

Here its the choice between a for/do loop, and sparse matrices. Personally, I like the abstraction of constructing a discrete operator to approximate my derivatives

Boundary conditions We have $v_0^n = v_{N+1}^n = 0$. Note choice of x grid in code. And how to change to periodic conditions.

Brownian motion Random walk / Brownian motion, the physical implementation of the heat/diffusion equation. [m12_random_walk.m]

We move back-and-forth between discrete and continuous!

```
DISCRETE                      CONTINUOUS
--------                      ----------
bouncing molecules            continuum models of physics
finite-difference approx      PDE
random walk                   Brownian motion
dots on computer screen       our perception of movie
floating-point arithmetic     real arithmetic
```

Note that these are all happening on different scales, which need not be tied to one another. Notably the PDE is a continuum model of what molecules do. We then discretize this to solve a discrete problem on a grid on the computer, but this grid has little (or nothing) to do with the molecular scale.

Nonlinear PDE

The heat equation was linear, dating to Fourier in 1807, and it can be more or less solved analytically.

Let's consider now nonlinear PDE, intractable analytically but easy to solve numerically.

Fisher-KPP Equation Independent 1937 discoveries for biological applications (spread of species): Fisher; Kolmogorov, Petrovsky, and Piscounov

$$u_t = \epsilon u_{xx} + u - u^2.$$

Solutions: traveling waves. Similar more complicated behaviour: Hodgkin-Huxley eqs for neural impulses (1952).

Explicit finite difference model is similar to heat equation. Note we have "nonhomogeneous" BCs $u(0) = 1$, $u(20) = 0$; the former is implemented by using an extra vector "BC". [m13_fisher_kpp.m]

```
                +--                                                    --+
                |         |-2   1            |  |  n |     |1/h^2|       |
                |         |                  |  | v  |     |     |       |
                |         | 1  -2   1        |  |  1 |     |  0  |       |
                |         |                  |  |    |     |     |       |
 n+1    n       |         |   .   .   .      |  |    |     |  .  |     2 |
v    = v  + k * | eps/h^2 |    .   .   .     |  |    |  +  |  .  |  +  (v -v ) |
                |         |     .   .   .    |  |    |     |  .  |       j  j  |
                |         |                  |  |    |     |     |       |
                |         |       1   -2   1 |  |  n |     |     |       |
                |         |                  |  | v  |     |     |       |
                |         |            1  -2 |  |  N |     |  0  |       |
                +--                                                    --+
                                                          BC
```

Stability in finite difference calculations

A fourth-order problem

$$u_t = -u_{xxxx}.$$

How to discretize? Think $(u_{xx})_{xx}\ldots$, this leads, eventually, to

$$v_j^{n+1} = v_j^n - \frac{k}{h^4}\left(v_{j-2}^n - 4v_{j-1}^n + 6v_j^n - 4v_{j+1}^n + v_{j+2}^n\right)$$

or

$$\frac{d}{dt}v_j = \frac{1}{h^4}\left(v_{j-2} - 4v_{j-1} + 6v_j - 4v_{j+1} + v_{j+2}\right)$$

$$\frac{d}{dt}v = -Hv$$

[m14_biharmonic.m] Note ridiculously small time steps required. Let's try to see why (a stability issue) and what we can do about it (implicit A-stable ODE methods).

von Neumann analysis *[For your interest, will not be tested]* One approach is *von Neumann Analysis* of the finite difference formula, also known as *discrete Fourier analysis*, invented in the late 1940s.

Suppose we have periodic boundary conditions and that at step n we have a (complex) sine wave

$$v_j^n = \exp(i\xi x_j) = \exp(i\xi jh),$$

for some wave number ξ. Higher ξ is more oscillatory. We will analysis whether this wave grows in amplitude or decays (for each ξ). For stability, we want all waves to decay.

For the biharmonic diffusion equation, we substitute this wave into the finite difference scheme above, and factor out $\exp(i\xi h)$ to get

$$v_j^{n+1} = g(\xi)v_j^n,$$

with the *amplification factor*

$$g(\xi) = 1 - \frac{k}{h^4}\left(e^{-i2\xi h} - 4e^{-i\xi h} + 6 - 4e^{i\xi h} + e^{i2\xi h}\right).$$

This can be simplified to:

$$g(\xi) = 1 - \frac{16k}{h^4}\sin(\xi\, h/2).$$

As ξ ranges over various values sin is bounded by 1 so we have $1 - 16k/h^4 \le g(\xi) \le 1$.

A mode will blow up if $|g(\xi)| > 1$. Thus for stability we want to ensure $|g(\xi)| \le 1$ for all ξ , i.e.,

$$1 - 16k/h^4 \ge -1, \qquad \text{or} \qquad \boxed{k \le h^4/8.}$$

For $h = 0.025$, as in the Matlab code, this gives $\boxed{k \le 4.883e-08}$. This matches our experiment convincingly, but confirms that this finite difference formula is not really practical.

Method-of-lines

As an alternative to von Neumann analysis, we follow the linear stability analysis for the ODE methods. The spatial discretization gives us (numerically anyway) the eigenvalues of the *semidiscrete* system. Need these eigenvalues to lie inside the absolute stability region of the ODE method.

Note: this involves the eigenvalues of the semidiscrete system, not the original right-hand-side of the PDE.

Demo: in Matlab, run m14_biharmonic, then use eigs to compute 'largest magnitude' eigenvalues of the *discretized* biharmonic operator: need k times these less than 2 for forward Euler stability. Note this gives almost the same restriction as observed in practice (and calculated with von Neumann analysis)

Implicit methods for PDEs

Apply implicit (A stable) methods to the semidiscrete method-of-lines form. For example, let's look at the heat equation $u_t = \Delta u$. If we apply backward Euler:

$$\frac{v^{n+1} - v^n}{k} = Lv^{n+1},$$

(c.f., forward Euler which has Lv^n on the RHS.)

Recall backward Euler is A-stable: stable for all eigenvalues in the left-half plane.

Similarly, the biharmonic hyperdiffusion equation with a matrix H:

$$\frac{v^{n+1} - v^n}{k} = -Hv^{n+1}.$$

[m15_be_heat.m][m16_be_biharm.m]

Example: Kuramoto-Sivashinsky equation

$$u_t = -u_{xx} - u_{xxxx} - (u^2/2)_x.$$

Ignore nonlinearity and think about what each "diffusion" term does.

- Long waves grow because of $-u_{xx}$;
- short waves decay because of $-u_{xxxx}$;
- the nonlinear term transfers energy from long to short.

[m17_kuramoto_sivashinsky.m] Note stability and chaotic behaviour of solution.

We treat the linear stiff terms implicitly and the nonlinear (but hopefully nonstiff) terms explicitly—an "IMEX" (implicit/explicit) discretization.

Order of accuracy in PDE finite difference calculations

We looked at this for ODEs before. We apply similar ideas for the PDE.

[m18_be_accuracy.m] Note: we have errors from both h and k. We plot error against k and h (in Figures 2 and 3) but these may not (and here do not) expose the truncation of the spatial and temporal schemes separately.

Local Truncation Error (again, for PDEs now)

As for ODEs, substitute equation solution into the descrete method and see what is left over. Example backward Euler applied to heat equation:

$$\frac{v_j^{n+1} - v_j^n}{k} = \frac{v_{j+1}^{n+1} - 2v_j^{n+1} + v_{j-1}^{n+1}}{h^2}.$$

- Replace v_j^{n-1} by Taylor series for equiv. value of u.
- Cancel terms to find local truncation error.

We often (try to) separate h and k dependences and talk about, e.g., $O(k) + O(h^2)$ accuracy. First-order accurate in time and second-order accurate in space.

Example: backward Euler

(all functions evaluated at (x_j, t_n) unless otherwise stated).

Taylor:

$$u(x, t_{n+1}) = u + ku_t + k^2/2u_{tt} + k^3/6u_{ttt} + \dots$$

And 2D Taylor gives

$$u(x_{j\pm1}, t^{n+1}) = u \pm hu_x + ku_t + \frac{h^2}{2}u_{xx} + \frac{k^2}{2}u_{tt} \pm hku_{xt} \pm \frac{h^3}{6}u_{xxx} + \frac{k^3}{6}u_{ttt} \pm \frac{k^2h}{2}u_{ttx} + \frac{kh^2}{2}u_{txx} + \frac{h^4}{24}u_{xxxx} + \dots$$

Now compute the LHS and RHS of the backward Euler method. Then we have:

$$LHS - RHS = u_t - u_{xx} + k/2u_{tt} + h^2/12u_{xxxx} + \dots$$

But $u_t = u_{xx}$ so we have the local truncation error is

$$O(k) + O(h^2)$$

(advanced: there is some subtly about h and k: we assumed $k = O(h)$ in this derivation...)

Forward Euler

Forward Euler is also $O(k) + O(h^2)$.

[m19_fe_accuracy.m] These results are less clear. Vary the $k = 0.25h^2$ parameter...

Why does this seem to give 2nd-order (in h)? Discuss w.r.t. stability. $k = O(h^2)$ Maybe you only want first-order accuracy, is so, this extra work is wasteful.

(Yet another "definition" of stiffness: if your choice of timestep k is motivated by stability rather than accuracy, you are probably dealing with a stiff problem.)

Higher-order in time

Even if we want second-order perhaps there are better ways, use a better ODE solve: trapezoidal/trapezium rule in time + second order in space. When used on heat equation, this is called "Crank–Nicolson":

$$v^{n+1} = v^n + \frac{k}{2} L v^{n+1} + \frac{k}{2} L v^n.$$

or you can write this as

$$B v^{n+1} = A v^n$$

[m20_cn_accuracy.m] And note we observe 2nd-order clearly in space and time with $k = O(h)$.

Caution Sometimes hard to tell from numerical convergence study which terms are dominating. Can also design tests to isolate the error components in h and k.

Lecture 6: PDEs in higher dimensions

Advection

1D: $u_t + a u_x = 0$.

In 2D:

$$u_t + a(x,y) u_x + b(x,y) u_y = 0.$$

Or more generally, we can write this as:

$$u_t + \nabla \cdot (\vec{w} u) = 0,$$

with a vector field $\vec{w}(x,y)$.

[m25_2d_adv.m]

Advection and the wave equation are quite different from diffusion: they are hyperbolic and "information" about the solution travels along characteristics. These are the lines traced out my the vector field $w(x,y)$.

The numerics are a bit different too: this code uses "upwinding" finite differences which are appropriate for advection-dominited problems, but we haven't talked about them in this course.

But we could look more carefully about constructing the matrices in this code...

Heat equation

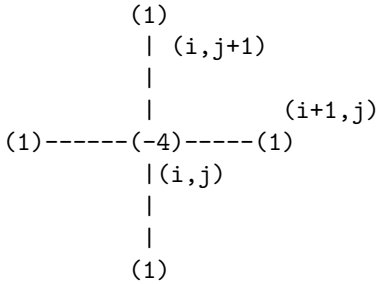
$$u_t = \nabla^2 u = u_{xx} + u_{yy}$$

on a square or rectangle. We can apply centered 2nd-order approximation to each derivative.

In the method of lines approach, we write

$$u_{xx} + u_{yy} \approx \frac{v^n_{i-1,j} - 2v^n_{ij} + v^n_{i+1,j}}{h^2} + \frac{v^n_{i,j-1} - 2v^n_{ij} + v^n_{i,j+1}}{h^2}.$$

This gives a stencil in *space* (then still need to deal with time).



Using forward or backward Euler, accuracy is $O(k + h^2)$.

And a stability restriction for FE of $k < h^2/4$.

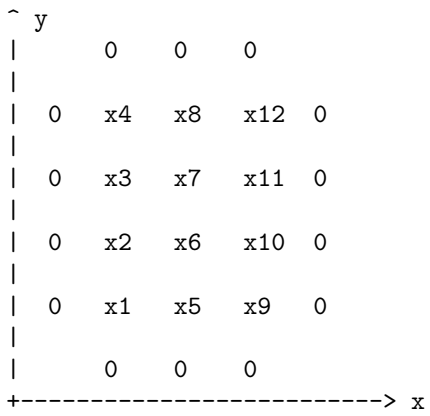
In principle, our "finite difference Laplacian" maps a matrix of 2D grid data to another such, and is thus a "4D tensor". However, in practice we stretch out 2D to 1D, so that the tensor becomes a matrix:

$$\frac{v}{dt} = Lv.$$

How does this "stretch" work? It defines an ordering of the grid points. In Matlab: meshgrid() and (:), see later.

Matrix structure

Let's look at the structure of L. We choose an ordering for the grid points (why this one? see below) and assume zero boundary conditions:



with corresponding unknowns $v_1, \ldots, v_{12}$. The discrete Laplacian now looks like this:

$$Lv = \frac{1}{h^2} \begin{bmatrix} -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 \end{bmatrix} \begin{bmatrix} v1 \\ v2 \\ v3 \\ v4 \\ v5 \\ v6 \\ v7 \\ v8 \\ v9 \\ v10 \\ v11 \\ v12 \end{bmatrix}.$$

Lecture 7: Application to Image Processing

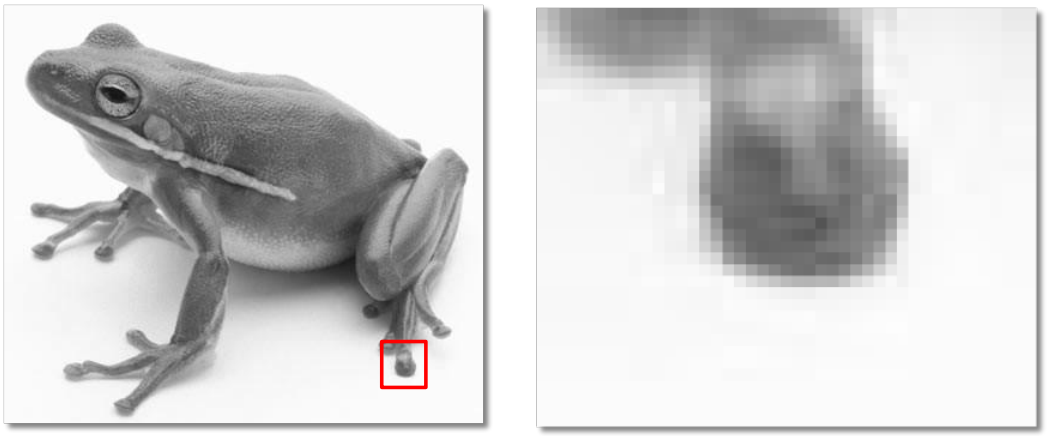


Figure 1: Digital image $u_{i,j}$ is a discretised and quantised version of "real" scene $u(x, y)$

A grey-scale image is simply a two dimensional $N \times M$ array of pixel values $u_{i,j}$, where $i \in 1 \dots L_x$ and $j \in 1 \dots L_y$. However, we can also consider that $u_{i,j}$ are samples of some continuous function $u(x, y)$ defined on the domain $(x, y) \in \Omega \subset \mathbb{R}^+$, where $\mathbb{R}^+$ is the set of positive real numbers. For the sake of simplicity, we set the domain so that $\Omega = \{x, y \in \mathbb{R}^+ : x < L_x \wedge y < L_y\}$, so that the distances between neighbouring pixels will be equal to 1.

In this case, an image consists of samples from this continuous function

$$u_{ij} = u(i, j)$$

We can consider a "real" image to consist of both a source term u, which we are trying to recover, and an additive noise term n

$$u_{i,j} = u(i, j) + n(i, j)$$

Sobol Edge Filter

How could you find edges in an image. . . .?

The gradient of u provides edge information. When magnitude of $\frac{\partial u}{\partial x}$ is high, likely we are at an edge

$$\left(\frac{\partial u}{\partial x}\right)_i \approx D_c^x u_i = u_{i+1} - u_{i-1}$$

Note we are defining an operator D_c^x that applies the central finite difference formula to u_i

For image processing, u_i values are normally corrupted by noise, which can affect the gradient. We can pre-smooth by triangle filter, e.g.

$$\left(\frac{\partial u}{\partial x}\right)_{i,j} \approx D_c^x (u_{i,j+1} + 2u_{i,j} + u_{i,j-1})$$

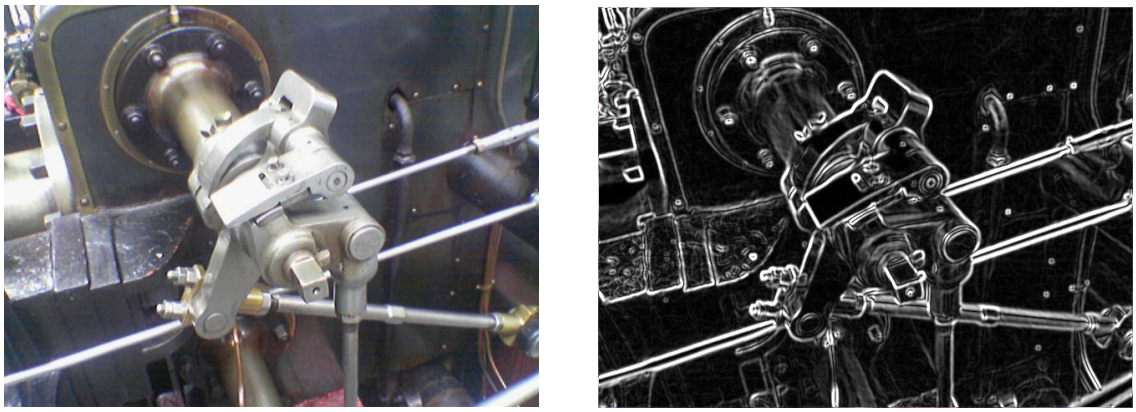


Figure 2: Sobol Edge Detect

We can extend this to a 2D image by taking the magnitude of the (two-dimensional) gradient

$$|\nabla u|_{i,j} \approx \sqrt{\left(\frac{\partial u}{\partial x}\right)^2_{i,j} + \left(\frac{\partial u}{\partial y}\right)^2_{i,j}}$$

Gaussian Blur Noise Reduction

We can reduce the amount of noise in an image by applying the time-dependent heat equation. Noise has a higher curvature than the image, so will be eliminated first

2D Heat Equation

The two-dimensional version of the heat equation is given as

$$u_t(x,y,t) = D(u_{xx}(x,y,t) + u_{yy}(x,y,t)) \tag{1}$$
$$u(x,y,0) = u_0(x,y) \tag{2}$$

As before, we will approximate the solution to the heat equation using finite differences, using a forward time, central space method.

The time derivative is unchanged from 1 dimension:

$$u_t(x_i, y_j, t_n) \approx \frac{u^{n+1}_{i,j} - u^n_{i,j}}{\Delta t}$$

where $t_n = \Delta t n$ for the set of positive integers $n = 0, \ldots, N$, $x_i = i$ for the set of positive integers $i = 1, \ldots, L$ and $y_j = j$ for the set of positive integers $j = 1, \ldots, L$.

We approximate u_{xx} and u_{yy} using a central difference

$$u_{xx}(x_i, y_j, t_n) \approx \frac{1}{\Delta x^2}(u^n_{i+1,j} - 2u^n_{i,j} + u^n_{i-1,j}) = u^n_{i+1,j} - 2u^n_{i,j} + u^n_{i-1,j}$$
$$u_{yy}(x_i, y_j, t_n) \approx \frac{1}{\Delta y^2}(u^n_{i,j+1} - 2u^n_{i,j} + u^n_{i,j-1}) = u^n_{i,j+1} - 2u^n_{i,j} + u^n_{i,j-1}$$

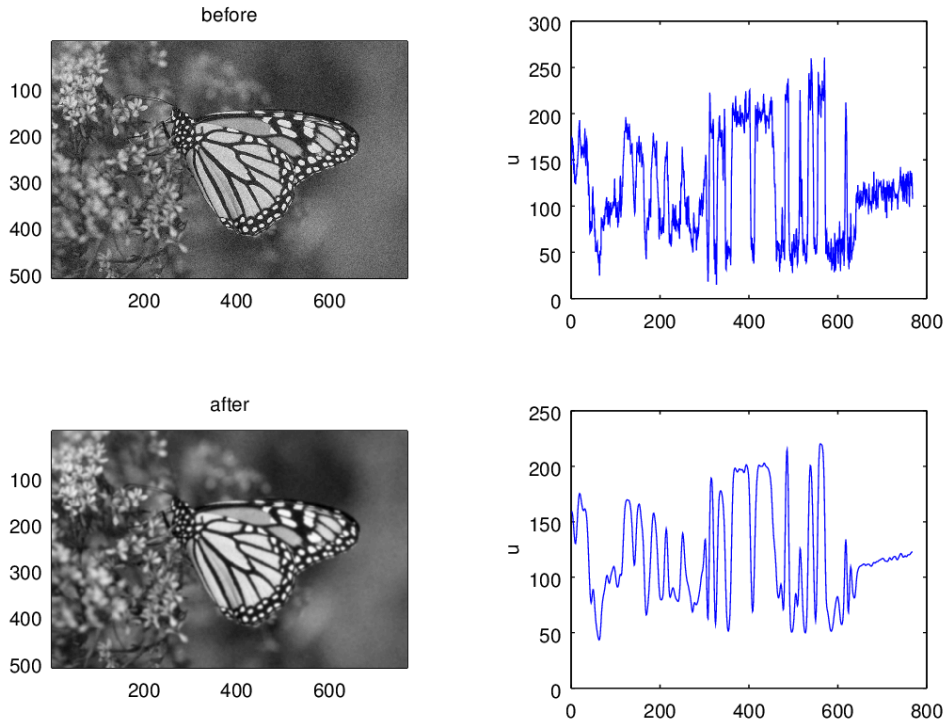


Figure 3: (left) butterfly image, top shows noisy image, bottom shows image after guassian blur (right) plot showing image values through a slice near the centre of the 2D image, top shows values from noisy image, bottom shows values after guassian blur

Putting these into the heat equation gives

$$\frac{u_{i,j}^{n+1} - u_{i,j}^n}{\Delta t} = D(u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - 4u_{i,j}^n) \tag{3}$$

$$u_i^{n+1} = u_i^n + \Delta t D(u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - 4u_{i,j}^n) \tag{4}$$

$$u_i^{n+1} = \Delta t D(u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n) + (1 - 4\Delta t D)u_{i,j}^n \tag{5}$$

Spatially Varying Diffusion

The heat equation $u_t = K\nabla u$ involves a diffusion constant K. We could replace that constant with a function of space

$$u_t = \nabla \cdot (K(\mathbf{x})\nabla u). \tag{6}$$

In one dimension, this would be

$$u_t = (K(x)u_x)_x. \tag{7}$$

A common approach to discretizing this is to use a *forward difference* $D_+^x \approx u_x$ on the u_x term

$$u_x \approx D_+^x u = \frac{u_{i+1} - u_i}{\Delta x}, \tag{8}$$

then evaluate $K(x)$ at the midpoint: $K(x_{i+\frac{1}{2}})$, so that

$$K(x)u_x \approx K(x_{i+\frac{1}{2}})\frac{u_{i+1} - u_i}{\Delta x}, \tag{9}$$

Then, differentiate the result approximately with a *backwards difference* D_-^x.

$$u_x \approx D_-^x u = \frac{u_i - u_{i-1}}{\Delta x}, \tag{10}$$

so that

$$(K(x)u_x)_x \approx D_-^x(K(x_{i+\frac{1}{2}})D_+^x u) = \frac{K(x_{i+\frac{1}{2}})\frac{u_{i+1}-u_i}{\Delta x} - K(x_{i-\frac{1}{2}})\frac{u_i - u_{i-1}}{\Delta x}}{\Delta x} \tag{11}$$

$$\approx \frac{K(x_{i+\frac{1}{2}})u_{i+1} - (K(x_{i+\frac{1}{2}}) + K(x_{i-\frac{1}{2}}))u_i + K(x_{i-\frac{1}{2}})u_{i-1}}{\Delta x^2} \tag{12}$$

Naturally, for constant diffusion $K(x) = K$ this reduces to

$$(K(x)u_x)_x \approx K\frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2} \tag{13}$$

For two dimensions, the spatially varying heat equation is

$$u_t = (K(x,y)u_x)_x + (K(x,y)u_y)_y. \tag{14}$$

and this can be discretized in a similar fashion to give

$$(K(x,y)u_x)_x \approx \frac{K(x_{i+\frac{1}{2}},y_j)u_{i+1,j} - (K(x_{i+\frac{1}{2}},y_j) + K(x_{i-\frac{1}{2}},y_j))u_{i,j} + K(x_{i-\frac{1}{2}},y_j)u_{i-1,j}}{\Delta x^2} \tag{15}$$

$$(K(x,y)u_y)_y \approx \frac{K(x_i,y_{j+\frac{1}{2}})u_{i,j+1} - (K(x_i,y_{j+\frac{1}{2}}) + K(x_i,y_{j-\frac{1}{2}}))u_{i,j} + K(x_i,y_{j-\frac{1}{2}})u_{i,j-1}}{\Delta y^2} \tag{16}$$

$$u_t \approx \frac{u^{n+1} - u^n}{\Delta t}. \tag{17}$$

If only the nodal values for $K(x,y)$ are known, then a reasonable approximation is to use the average of two neighbouring grid points

$$K(x_i,y_{j+\frac{1}{2}}) = \frac{1}{2}(K_{i,j+1} + K_{i,j}) \tag{18}$$

$$K(x_{i+\frac{1}{2}},y_j) = \frac{1}{2}(K_{i+1,j} + K_{i,j}) \tag{19}$$

Which results in

$$(K(x,y)u_x)_x \approx \frac{(K_{i+1,j} + K_{i,j})u_{i+1,j} - ((K_{i+1,j} + 2K_{i,j} + K_{i-1,j}))u_{i,j} + (K_{i-1,j} + K_{i,j})u_{i-1,j}}{2\Delta x^2} \tag{20}$$

$$(K(x,y)u_y)_y \approx \frac{(K_{i,j+1} + K_{i,j})u_{i,j+1} - ((K_{i,j+1} + 2K_{i,j} + K_{i,j-1}))u_{i,j} + (K_{i,j-1} + K_{i,j})u_{i,j-1}}{2\Delta y^2} \tag{21}$$

$$u_t \approx \frac{u^{n+1} - u^n}{\Delta t}. \tag{22}$$

For $\Delta x = \Delta y = 1$ this simplifies to

$$u_{i,j}^{n+1} = u_{i,j}^n + \frac{1}{2}\Delta t(\tag{23}$$

$$(K_{i+1,j} + K_{i,j})u_{i+1,j} + (K_{i-1,j} + K_{i,j})u_{i-1,j} \tag{24}$$

$$(K_{i,j+1} + K_{i,j})u_{i,j+1} + (K_{i,j-1} + K_{i,j})u_{i,j-1} \tag{25}$$

$$- (K_{i+1,j} + K_{i-1,j} + K_{i,j+1} + K_{i,j-1} + 4K_{i,j})u_{i,j}) \tag{26}$$

What shall we use for $K(x,y)$? For noise reduction in images, we may want to preserve image features, or edges. We know that edges are associated with areas of high $|\nabla u|$. We can therefore set $K(x,y) = g(|\nabla u|)$, where g is a given function. Perona & Malik proposed using $g(s) = 1/(1 + \frac{s^2}{\lambda^2})$. Recall that,

$$|\nabla u|_{i,j} \approx \sqrt{\left(\frac{\partial u}{\partial x}\right)^2_{i,j} + \left(\frac{\partial u}{\partial y}\right)^2_{i,j}}$$

therefore we can set

$$\begin{aligned} s^2_{i,j} &= (D^x_c u^n_{i,j})^2 + (D^y_c u^n_{i,j})^2 \\ &= (u^n_{i+1,j} - u^n_{i-1,j})^2 + (u^n_{i,j+1} - u^n_{i,j-1})^2 \end{aligned}$$

and

$$K_{i,j} = \frac{1}{1 + \frac{s^2_{i,j}}{\lambda^2}}$$