Unit 3: Numerical Calculus (Part 1)

Motivation

Since the time of Newton, calculus has been ubiquitous in science

Many (most?) calculus problems that arise in applications do not have closed-form solutions

Numerical approximation is essential!

Epitomizes idea of Scientific Computing as developing and applying numerical algorithms to problems of continuous mathematics

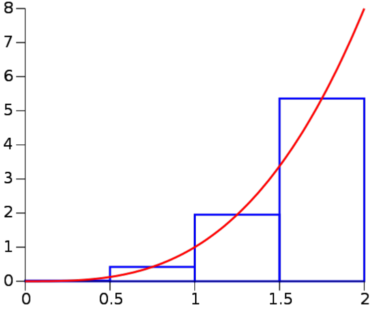
In this Unit we will consider:

- Numerical integration
- Numerical differentiation
- Numerical methods for ordinary differential equations
- Numerical methods for partial differential equations

Integration

Approximating a definite integral using a numerical method is called quadrature

The familiar Riemann sum idea suggests how to perform quadrature



We will examine more accurate/efficient quadrature methods

Integration

Question: Why is quadrature important?

We know how to evaluate many integrals analytically, *e.g.*

$$\int_0^1 e^x \mathrm{d}x \qquad \text{or} \qquad \int_0^\pi \cos x \mathrm{d}x$$

But how about $\int_1^{2000} \exp(\sin(\cos(\sinh(\cosh(\tan^{-1}(\log(x))))))) \mathrm{d}x$?

Integration

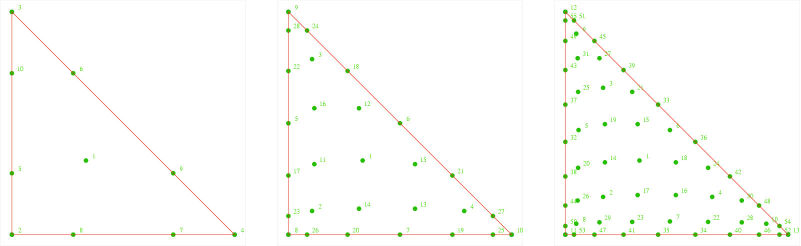
We can numerically approximate this integral in Python using quadrature

```
Python 3.8.6 (default, Sep 28 2020, 04:41:02)
[Clang 11.0.3 (clang-1103.0.32.62)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import scipy.integrate as spi
>>> from math import *
>>> def f(x):
>>> def f(x):
...     return exp(sin(cos(sinh(cosh(atan(log(x)))))))
>>> spi.quad(f,1,2000)
(1514.7806778270258, 4.231109728875231e-06)
```

Integration

Quadrature also generalizes naturally to higher dimensions, and allows us to compute integrals on irregular domains

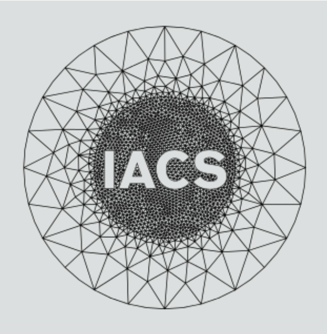
For example, we can approximate an integral on a triangle based on a finite sum of samples at quadrature points



Three different quadrature rules on a triangle

Integration

Can then evaluate integrals on complicated regions by "triangulating" (AKA "meshing")



Differentiation

Numerical differentiation is another fundamental tool in Scientific Computing

We have already discussed the most common, intuitive approach to numerical differentiation: finite differences

$$
\begin{aligned}
f'(x) &= \frac{f(x+h)-f(x)}{h} + O(h) && \text{(forward difference)} \\
f'(x) &= \frac{f(x)-f(x-h)}{h} + O(h) && \text{(backward difference)} \\
f'(x) &= \frac{f(x+h)-f(x-h)}{2h} + O(h^2) && \text{(centered difference)} \\
f''(x) &= \frac{f(x+h)-2f(x)+f(x-h)}{h^2} + O(h^2) && \text{(centered, 2nd deriv.)} \\
&\vdots
\end{aligned}
$$

Differentiation

We will see how to derive these and other finite difference formulas and quantify their accuracy

Wide range of choices, with trade-offs in terms of

- accuracy
- stability
- complexity

Differentiation

We saw at the start of the course that finite differences can be sensitive to rounding error when h is "too small"

But in most applications we obtain sufficient accuracy with h large enough that rounding error is still negligible[1]

Hence finite differences generally work very well, and provide a very popular approach to solving problems involving derivatives

[1] That is, h is large enough so that rounding error is dominated by discretization error

ODEs

The most common situation in which we need to approximate derivatives is in order to solve differential equations

Ordinary Differential Equations (ODEs): Differential equations involving functions of one variable

Some example ODEs:

- $y'(t) = y^2(t) + t^4 - 6t$, $y(0) = y_0$ is a first order Initial Value Problem (IVP) ODE
- $y''(x) + 2xy(x) = 1$, $y(0) = y(1) = 0$ is a second order Boundary Value Problem (BVP) ODE

ODEs: IVP

A familiar IVP ODE is Newton's Second Law of Motion: suppose position of a particle at time $t \geq 0$ is $y(t) \in \mathbb{R}$

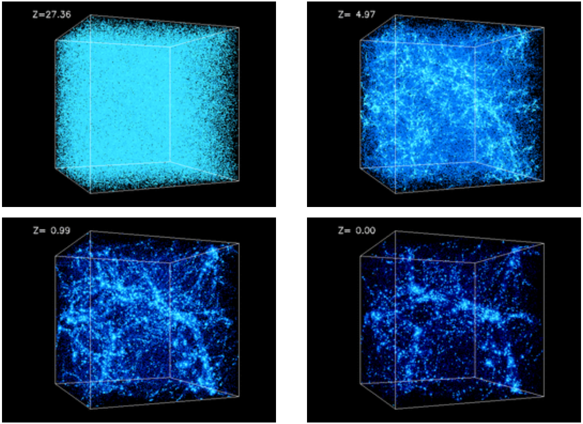
$$y''(t) = \frac{F(t, y, y')}{m}, \qquad y(0) = y_0, y'(0) = v_0$$

This is a scalar ODE ($y(t) \in \mathbb{R}$), but it's common to simulate a system of N interacting particles

e.g. F could be gravitational force due to other particles, then force on particle i depends on positions of the other particles

ODEs: IVP

N-body problems are the basis of many cosmological simulations:
Recall galaxy formation simulations from Unit 0



Computationally expensive when N is large!

ODEs: BVP

ODE boundary value problems are also important in many circumstances

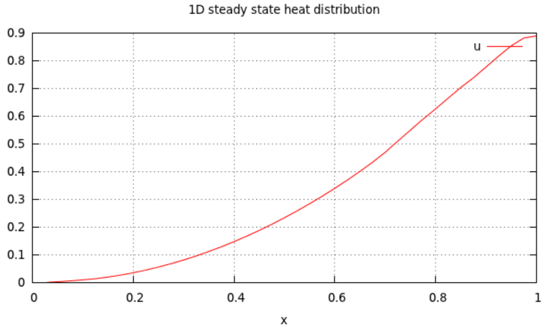
For example, steady state heat distribution in a "1D rod"

Apply heat source $f(x) = x^2$, impose "zero" temperature at $x = 0$, insulate at $x = 1$:

$$-u''(x) = x^2, \quad u(0) = 0, u'(1) = 0$$

ODEs: BVP

We can approximate via finite differences: use F.D. formula for $u''(x)$



PDEs

It is also natural to introduce time-dependence for the temperature in the "1D rod" from above

Hence now u is a function of x and t, so derivatives of u are partial derivatives, and we obtain a partial differential equation (PDE)

For example, the time-dependent heat equation for the 1D rod is given by:

$$\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} = x^2, \quad u(x,0) = 0, u(0,t) = 0, \frac{\partial u}{\partial x}(1,t) = 0$$

This is an Initial-Boundary Value Problem (IBVP)

PDEs

Also, when we are modeling continua[2] we generally also need to be able to handle 2D and 3D domains

e.g. 3D analogue of time-dependent heat equation on a domain $\Omega \subset \mathbb{R}^3$ is

$$\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} - \frac{\partial^2 u}{\partial z^2} = f(x,y,z), \quad u = 0 \text{ on } \partial\Omega$$

[2] *e.g.* temperature distribution, fluid velocity, electromagnetic fields, ...

PDEs

This equation is typically written as

$$\frac{\partial u}{\partial t} - \Delta u = f(x, y, z), \quad u = 0 \text{ on } \partial\Omega$$

where $\Delta u \equiv \nabla \cdot \nabla u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}$

Here we have:

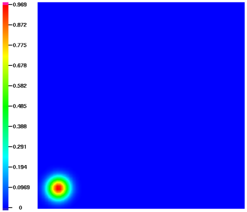
- The Laplacian, $\Delta \equiv \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$
- The gradient, $\nabla \equiv (\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z})$

PDEs

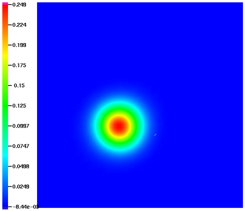
Can add a "transport" term to the heat equation to obtain the convection-diffusion equation, *e.g.* in 2D we have

$$\frac{\partial u}{\partial t} + (w_1(x,y), w_2(x,y)) \cdot \nabla u - \Delta u = f(x,y), \quad u = 0 \text{ on } \partial\Omega$$

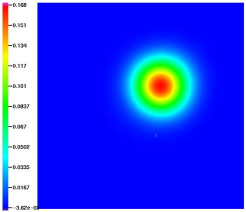
$u(x,t)$ models concentration of some substance, *e.g.* pollution in a river with current (w_1, w_2)



$t = 0$



$t = 3$

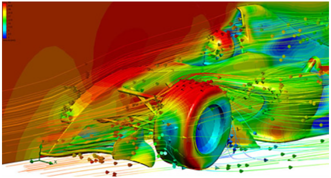


$t = 5$

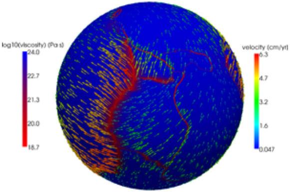
PDEs

Numerical methods for PDEs are a major topic in scientific computing

Recall examples from Unit 0:



CFD



Geophysics

The finite difference method is an effective approach for a wide range of problems, hence we focus on F.D. in AM205[3]

[3] There are many important alternatives, *e.g.* finite element method, finite volume method, spectral methods, boundary element methods . . .

Summary

Numerical calculus encompasses a wide range of important topics in scientific computing!

As always, we will pay attention to stability, accuracy and efficiency of the algorithms that we consider

Quadrature

Suppose we want to evaluate the integral $I(f) \equiv \int_a^b f(x)\mathrm{d}x$

We can proceed as follows:

1. Approximate f using a polynomial interpolant p_n
2. Evaluate $Q_n(f) \equiv \int_a^b p_n(x)\mathrm{d}x$, since we know how to integrate polynomials

$Q_n(f)$ provides a quadrature formula, and we should have $Q_n(f) \approx I(f)$

A quadrature rule based on an interpolant p_n at $n+1$ equally spaced points in $[a, b]$ is known as Newton–Cotes formula of order n

Newton–Cotes Quadrature

Let $x_k = a + kh$, $k = 0, 1, \ldots, n$, where $h = (b - a)/n$

We write the interpolant of f in Lagrange form as

$$p_n(x) = \sum_{k=0}^{n} f(x_k) L_k(x), \quad \text{where} \quad L_k(x) \equiv \prod_{i=0, i\neq k}^{n} \frac{x - x_i}{x_k - x_i}$$

Then

$$Q_n(f) = \int_a^b p_n(x)\mathrm{d}x = \sum_{k=0}^{n} f(x_k) \int_a^b L_k(x)\mathrm{d}x = \sum_{k=0}^{n} w_k f(x_k)$$

where $w_k \equiv \int_a^b L_k(x)\mathrm{d}x \in \mathbb{R}$ is the kth quadrature weight

Newton–Cotes Quadrature

Note that quadrature weights do not depend on f, hence can be precomputed and stored

$n = 1 \implies$ Trapezoid rule (See lecture)

$n = 2 \implies Q_2(f) = \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right]$ Simpson rule

We can also develop higher-order Newton–Cotes formulae in the same way

Error Estimates

Let $E_n(f) \equiv I(f) - Q_n(f)$

Then

$$
\begin{aligned}
E_n(f) &= \int_a^b f(x)\mathrm{d}x - \sum_{k=0}^{n} w_k f(x_k) \\
&= \int_a^b f(x)\mathrm{d}x - \sum_{k=0}^{n} \left(\int_a^b L_k(x)\mathrm{d}x \right) f(x_k) \\
&= \int_a^b f(x)\mathrm{d}x - \int_a^b \left(\sum_{k=0}^{n} L_k(x) f(x_k) \right) \mathrm{d}x \\
&= \int_a^b f(x)\mathrm{d}x - \int_a^b p_n(x)\mathrm{d}x \\
&= \int_a^b \left(f(x) - p_n(x) \right) \mathrm{d}x
\end{aligned}
$$

And we have an expression for $f(x) - p_n(x)$

Error Estimates

Recall from Unit I

$$f(x) - p_n(x) = \frac{f^{n+1}(\theta)}{(n+1)!}(x - x_0)\dots(x - x_n)$$

Hence

$$|E_n(f)| \leq \frac{M_{n+1}}{(n+1)!}\int_a^b |(x - x_0)(x - x_1)\cdots(x - x_n)|\mathrm{d}x$$

where $M_{n+1} = \max_{\theta \in [a,b]} |f^{n+1}(\theta)|$

Error Estimates

See lecture: Trapezoid rule error bound

$$|E_1(f)| \leq \frac{(b-a)^3}{12} M_2$$

The bound for E_n depends directly on the integrand f (via M_{n+1})

Just like with the Lebesgue constant, it is informative to be able to compare quadrature rules independently of the integrand

Error Estimates: Another Perspective

Theorem: If Q_n integrates polynomials of degree n exactly, then $\exists C_n > 0$ such that $|E_n(f)| \leq C_n \min_{p \in \mathbb{P}_n} \|f - p\|_\infty$

Proof: For $p \in \mathbb{P}_n$, we have

$$
\begin{aligned}
|I(f) - Q_n(f)| &\leq |I(f) - I(p)| + |I(p) - Q_n(f)| \\
&= |I(f - p)| + |Q_n(f - p)| \\
&\leq \int_a^b \mathrm{d}x \|f - p\|_\infty + \left(\sum_{k=0}^{n} |w_k| \right) \|f - p\|_\infty \\
&\equiv C_n \|f - p\|_\infty
\end{aligned}
$$

where

$$
C_n \equiv b - a + \sum_{k=0}^{n} |w_k|
$$

Error Estimates

Hence a convenient way to compare accuracy of quadrature rules is to compare the polynomial degree they integrate exactly

Newton–Cotes of order n is based on polynomial interpolation, hence in general integrates polynomials of degree n exactly[4]

[4] Also follows from the M_{n+1} term in the error bound

Runge's Phenomenon Again . . .

But Newton–Cotes formulae are based on interpolation at equally spaced points

Hence they're susceptible to Runge's phenomenon, and we expect them to be inaccurate for large n

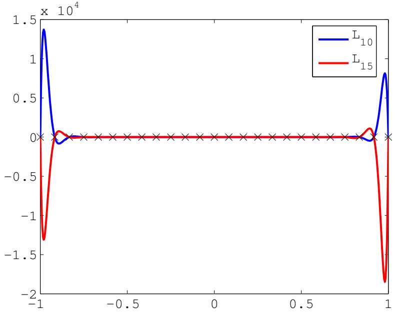
Question: How does this show up in our bound

$$|E_n(f)| \leq C_n \min_{p \in \mathbb{P}_n} \|f - p\|_\infty \quad ?$$

Runge Phenomenon Again ...

Answer: In the constant C_n

Recall that $C_n \equiv b - a + \sum_{k=0}^{n} |w_k|$, and that $w_k \equiv \int_a^b L_k(x)\mathrm{d}x$



If the L_k "blow up" due to equally spaced points, then C_n can also "blow up"

Runge Phenomenon Again ...

In fact, we know that $\sum_{k=0}^{n} w_k = b - a$, why?

This tells us that if all the w_k are positive, then

$$C_n = b - a + \sum_{k=0}^{n} |w_k| = b - a + \sum_{k=0}^{n} w_k = 2(b - a)$$

Hence positive weights $\implies$ C_n is a constant, independent of n and hence $Q_n(f) \to I(f)$ as $n \to \infty$

Runge Phenomenon Again...

But with Newton–Cotes, quadrature weights become negative for $n > 8$ (*e.g.* in example above $L_{15}(x)$ would clearly yield $w_{15} < 0$)

Key point: Newton–Cotes is not useful for large n

However, there are two natural ways to get quadrature rules that converge as $n \to \infty$:

- Integrate piecewise polynomial interpolant
- Don't use equally spaced interpolation points

We consider piecewise polynomial-based quadrature rules first

Composite Quadrature Rules

Integrating piecewise polynomial interpolant $\Longrightarrow$ composite quadrature rule

Suppose we divide $[a, b]$ into m subintervals, each of width $h = (b - a)/m$, and $x_i = a + ih$, $i = 0, 1, \ldots, m$

Then we have:

$$I(f) = \int_a^b f(x)\mathrm{d}x = \sum_{i=1}^{m} \int_{x_{i-1}}^{x_i} f(x)\mathrm{d}x$$

Composite Trapezoid Rule

Composite trapezoid rule: Apply trapezoid rule to each interval, *i.e.* $\int_{x_{i-1}}^{x_i} f(x)\mathrm{d}x \approx \frac{1}{2}h[f(x_{i-1}) + f(x_i)]$

Hence,

$$
\begin{aligned}
Q_{1,h}(f) &\equiv \sum_{i=1}^{m} \frac{1}{2}h[f(x_{i-1}) + f(x_i)] \\
&= h\left[\frac{1}{2}f(x_0) + f(x_1) + \cdots + f(x_{m-1}) + \frac{1}{2}f(x_m)\right]
\end{aligned}
$$

Composite Trapezoid Rule

Composite trapezoid rule error analysis:

$$
\begin{aligned}
E_{1,h}(f) &\equiv I(f) - Q_{1,h}(f) \\
&= \sum_{i=1}^{m} \left[\int_{x_{i-1}}^{x_i} f(x)\mathrm{d}x - \frac{1}{2}h[f(x_{i-1}) + f(x_i)] \right]
\end{aligned}
$$

Hence,

$$
\begin{aligned}
|E_{1,h}(f)| &\leq \sum_{i=1}^{m} \left| \int_{x_{i-1}}^{x_i} f(x)\mathrm{d}x - \frac{1}{2}h[f(x_{i-1}) + f(x_i)] \right| \\
&\leq \frac{h^3}{12} \sum_{i=1}^{m} \max_{\theta \in [x_{i-1}, x_i]} |f''(\theta)| \\
&\leq \frac{h^3}{12} m \|f''\|_\infty \\
&= \frac{h^2}{12}(b-a)\|f''\|_\infty
\end{aligned}
$$

Composite Simpson Rule

We can obtain the composite Simpson rule in the same way

Suppose that $[a, b]$ is divided into $2m$ intervals by the points $x_i = a + ih$, $i = 0, 1, \ldots, 2m$, $h = (b - a)/2m$

Applying Simpson rule on each interval[5] $[x_{2i-2}, x_{2i}]$, $i = 1, \ldots, m$ yields

$$\begin{aligned} Q_{2,h}(f) \quad &\equiv \quad \frac{h}{3}[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \cdots \\ &\qquad + 2f(x_{2m-2}) + 4f(x_{2m-1}) + f(x_{2m})] \end{aligned}$$

[*quadrat.py*] Demo of composite trapezoid rule and Simpson rule convergence

[5] Interval of width $2h$

Adaptive Quadrature

Composite quadrature rules are very flexible, *e.g.* we need not choose equally sized intervals

Intuitively, we should use smaller intervals where f varies rapidly, and larger intervals where f varies slowly

This can be achieved by adaptive quadrature:

1. Initialize to $m = 1$ (one interval)
2. On each interval, evaluate quadrature rule and estimate quadrature error
3. If error estimate $>$ TOL on interval i, subdivide to get two smaller intervals and return to step 2.

Question: How can we estimate the quadrature error on an interval?

Adaptive Quadrature

One straightforward way to estimate quadrature error on interval i is to compare to a more refined result for interval i

Let $I^i(f)$ and $Q_h^i(f)$ denote the exact integral and quadrature approximation on interval i, respectively

Let $\hat{Q}_h^i(f)$ denote a more refined quadrature approximation on interval i, *e.g.* obtained by subdividing interval i

Then for the error on interval i, we have:

$$|I^i(f) - Q_h^i(f)| \leq |I^i(f) - \hat{Q}_h^i(f)| + |\hat{Q}_h^i(f) - Q_h^i(f)|$$

Then, we suppose we can neglect $|I^i(f) - \hat{Q}_h^i(f)|$ so that we use $|\hat{Q}_h^i(f) - Q_h^i(f)|$ as a computable estimator for $|I^i(f) - Q_h^i(f)|$

Adaptive Quadrature

Python and MATLAB both have `quad` functions, although with different implementations. MATLAB's `quad` function implements an adaptive Simpson rule:

```
>> help quad
QUAD   Numerically evaluate integral, adaptive Simpson
quadrature.  Q = QUAD(FUN,A,B) tries to approximate the
integral of scalar-valued function FUN from A to B to
within an error of 1.e-6 using recursive adaptive Simpson
quadrature.
```

Next we consider the second approach to developing more accurate quadrature rules: unevenly spaced quadrature points

Gauss Quadrature

Recall that we can compare accuracy of quadrature rules based on the polynomial degree that is integrated exactly

So far, we haven't been very creative with our choice of quadrature points: Newton–Cotes $\Longleftrightarrow$ equally spaced

More accurate quadrature rules can be derived by choosing the x_i to maximize poly. degree that is integrated exactly

Resulting family of quadrature rules is called Gauss quadrature

Gauss Quadrature

Intuitively, with $n + 1$ quadrature points and $n + 1$ quadrature weights we have $2n + 2$ parameters to choose

Hence we might hope to integrate a poly. with $2n + 2$ parameters, *i.e.* of degree $2n + 1$

It can be shown that this is possible $\Longrightarrow$ Gauss quadrature

Again the idea is to integrate a polynomial interpolant, but we choose a specific set of interpolation points:

Gauss quad. points are roots of a Legendre polynomial[6]

[6]Adrien-Marie Legendre, 1752-1833, French mathematician

Gauss Quadrature

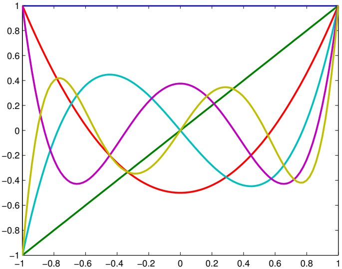
Briefly, Legendre polynomials $\{P_0, P_1, \ldots, P_n\}$ form an orthogonal basis for $\mathbb{P}_n$ in the "L^2 inner-product"

$$\int_{-1}^{1} P_m(x)P_n(x)\mathrm{d}x = \begin{cases} \frac{2}{2n+1}, & m = n \\ 0, & m \neq n \end{cases}$$

Gauss Quadrature

As with Chebyshev polys, Legendre polys satisfy a 3-term recurrence relation

$$
\begin{aligned}
P_0(x) &= 1 \\
P_1(x) &= x \\
(n+1)P_{n+1}(x) &= (2n+1)xP_n(x) - nP_{n-1}(x)
\end{aligned}
$$



The first six Legendre polynomials

Gauss Quadrature

Hence, can find the roots of $P_n(x)$ and derive the n-point Gauss quad. rule in the same way as for Newton–Cotes:

Integrate the Lagrange interpolant!

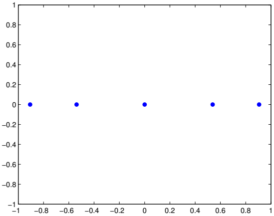
Gauss quadrature rules have been extensively tabulated for $x \in [-1, 1]$:

Number of points	Quadrature points	Quadrature weights
1	0	2
2	$-1/\sqrt{3}, 1/\sqrt{3}$	$1, 1$
3	$-\sqrt{3/5}, 0, \sqrt{3/5}$	$5/9, 8/9, 5/9$
$\vdots$	$\vdots$	$\vdots$

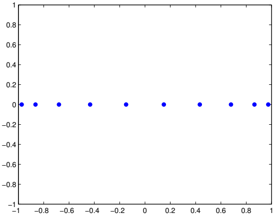
Key point: Gauss quadrature weights are always positive, hence Gauss quadrature converges as $n \to \infty$!

Gauss Quadrature Points

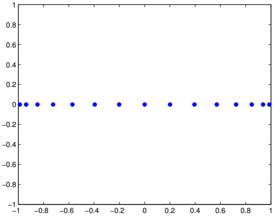
Points cluster toward ± 1, prevents Runge's phenomenon!



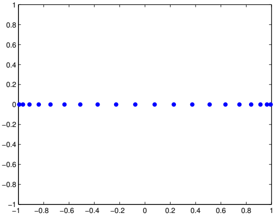
5 points



10 points



15 points



20 points

Generalization

Suppose we wish to evaluate exactly integrals of the form

$$\int_{-1}^{1} w(x) f(x)\, dx.$$

Then we can calculate quadrature based on polynomials u_k that are orthogonal with respect to the inner product

$$\langle u_j, u_k \rangle = \int_{-1}^{1} w(x) u_j(x) u_k(x) dx.$$

A typical example case is

$$w(x) = \frac{1}{\sqrt{1 - x^2}}.$$

Orthogonality relation is then

$$\langle u_j, u_k \rangle = \int_{-1}^{1} \frac{1}{\sqrt{1 - x^2}} u_j(x) u_k(x) dx.$$

Try the Chebyshev polynomials $u_j(x) = T_j(x) = \cos(j \cos^{-1} x)$.

Generalization

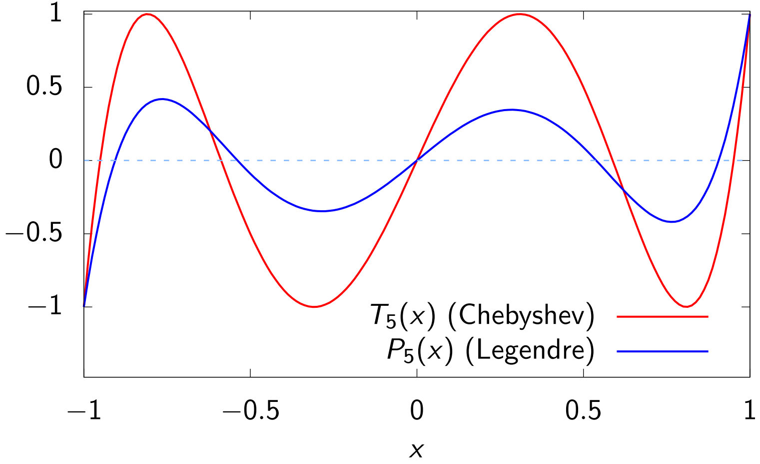
Using the substitution $x = \cos\theta$,

$$
\begin{aligned}
\langle T_j, T_k \rangle &= \int_{-1}^{1} \frac{1}{\sqrt{1-x^2}} \cos(j\cos^{-1}x)\cos(k\cos^{-1}x)dx \\
&= \int_0^{\pi} \frac{1}{\sqrt{1-\cos^2\theta}} \cos j\theta \cos k\theta (\sin\theta \, d\theta) \\
&= \int_0^{\pi} \cos j\theta \cos k\theta \, d\theta.
\end{aligned}
$$

Using the Fourier orthogonality relations, $\langle T_j, T_k \rangle = 0$ for $j \neq k$, so the Chebyshev polynomials are orthogonal with respect to this weight function.

Hence the roots of the Chebyshev polynomials can be used to construct a quadrature formula for this $w(x)$. This is just one example of many possible generalizations to Gauss quadrature.

Legendre/Chebyshev comparison



Chebyshev roots are closer to the ends—better sampling of the function near ± 1, as expected based on $w(x)$.

Gauss Quadrature

Python's `quad` function makes use of Clenshaw–Curtis quadrature, based on Chebyshev polynomials.

In MATLAB, `quadl` performs adaptive, composite Lobatto quadrature. Lobatto quadrature is closely related to Gauss quadrature, difference is that we ensure that -1 and 1 are quadrature points.

From `help quadl`:
" `QUAD may be most efficient for low accuracies with nonsmooth integrands.`
`QUADL may be more efficient than QUAD at higher accuracies with smooth integrands.` "

Take-away message: Gauss–Lobatto quadrature is usually more efficient for smooth integrands

Finite Difference Approximations

Given a function $f : \mathbb{R} \to \mathbb{R}$

We want to approximate derivatives of f via simple expressions involving samples of f

As we saw in Unit 0, convenient starting point is Taylor's theorem

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2}h^2 + \frac{f'''(x)}{6}h^3 + \cdots$$

Finite Difference Approximations

Solving for $f'(x)$ we get the forward difference formula

$$\begin{aligned} f'(x) &= \frac{f(x+h)-f(x)}{h} - \frac{f''(x)}{2}h + \cdots \\ &\approx \frac{f(x+h)-f(x)}{h} \end{aligned}$$

Here we neglected an $O(h)$ term

Finite Difference Approximations

Similarly, we have the Taylor series

$$f(x-h) = f(x) - f'(x)h + \frac{f''(x)}{2}h^2 - \frac{f'''(x)}{6}h^3 + \cdots$$

which yields the backward difference formula

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}$$

Again we neglected an $O(h)$ term

Finite Difference Approximations

Subtracting Taylor expansion for $f(x-h)$ from expansion for $f(x+h)$ gives the centered difference formula

$$
\begin{aligned}
f'(x) &= \frac{f(x+h) - f(x-h)}{2h} - \frac{f'''(x)}{6}h^2 + \cdots \\
&\approx \frac{f(x+h) - f(x-h)}{2h}
\end{aligned}
$$

In this case we neglected an $O(h^2)$ term

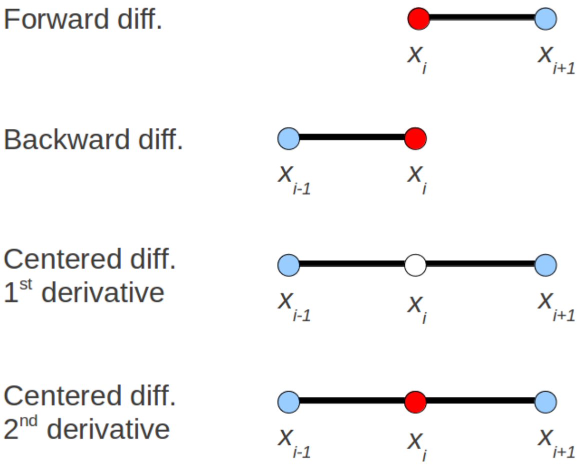
Finite Difference Approximations

Adding Taylor expansion for $f(x-h)$ and expansion for $f(x+h)$ gives the centered difference formula for the second derivative

$$
\begin{aligned}
f''(x) &= \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} - \frac{f^{(4)}(x)}{12}h^2 + \cdots \\
&\approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}
\end{aligned}
$$

Again we neglected an $O(h^2)$ term

Finite Difference Stencils



Finite Difference Approximations

We can use Taylor expansion to derive approximations with higher order accuracy, or for higher derivatives

This involves developing F.D. formulae with "wider stencils," *i.e.* based on samples at $x \pm 2h, x \pm 3h, \ldots$

But there is an alternative that generalizes more easily to higher order formulae:

Differentiate the interpolant!

Finite Difference Approximations

Linear interpolant at $\{(x_0, f(x_0)), (x_0 + h, f(x_0 + h))\}$ is

$$p_1(x) = f(x_0)\frac{x_0 + h - x}{h} + f(x_0 + h)\frac{x - x_0}{h}$$

Differentiating p_1 gives

$$p_1'(x) = \frac{f(x_0 + h) - f(x_0)}{h},$$

which is the forward difference formula

Question: How would we derive the backward difference formula based on interpolation?

Finite Difference Approximations

Similarly, quadratic interpolant, p_2, from interpolation points $\{x_0, x_1, x_2\}$ yields the centered difference formula for f' at x_1:

- Differentiate $p_2(x)$ to get a linear polynomial, $p_2'(x)$
- Evaluate $p_2'(x_1)$ to get centered difference formula for f'

Also, $p_2''(x)$ gives the centered difference formula for f''

Note: Can apply this approach to higher degree interpolants, and interp. pts. need not be evenly spaced

Finite Difference Approximations

So far we have talked about finite difference formulae to approximate $f'(x_i)$ at some specific point x_i

Question: What if we want to approximate $f'(x)$ on an interval $x \in [a, b]$?

Answer: We need to simultaneously approximate $f'(x_i)$ for x_i, $i = 1, \ldots, n$

Differentiation Matrices

We need a map from the vector $F \equiv [f(x_1), f(x_2), \ldots, f(x_n)] \in \mathbb{R}^n$ to the vector of derivatives $F' \equiv [f'(x_1), f'(x_2), \ldots, f'(x_n)] \in \mathbb{R}^n$

Let $\widetilde{F}'$ denote our finite difference approximation to the vector of derivatives, *i.e.* $\widetilde{F}' \approx F'$

Differentiation is a linear operator[7], hence we expect the map from F to $\widetilde{F}'$ to be an $n \times n$ matrix

This is indeed the case, and this map is a differentiation matrix, D

---

[7]Since $(\alpha f + \beta g)' = \alpha f' + \beta g'$

Differentiation Matrices

Row i of D corresponds to the finite difference formula for $f'(x_i)$, since then $D_{(i,:)}F \approx f'(x_i)$

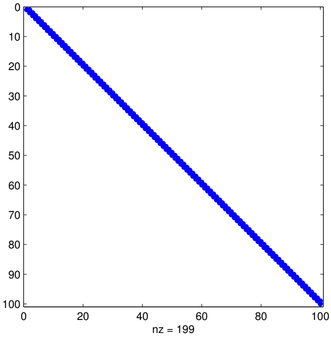
e.g. for forward difference approx. of f', non-zero entries of row i are

$$D_{ii} = -\frac{1}{h}, \quad D_{i,i+1} = \frac{1}{h}$$

This is a sparse matrix with two non-zero diagonals

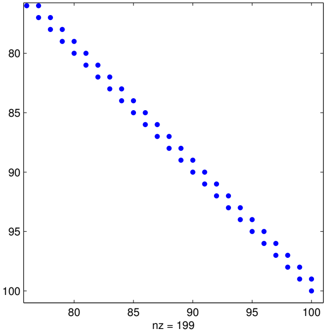
Differentiation Matrices

```
n=100
h=1/(n-1)
D=np.diag(-np.ones(n)/h)+np.diag(np.ones(n-1)/h,1)
plt.spy(D)
plt.show()
```



Differentiation Matrices

But what about the last row?

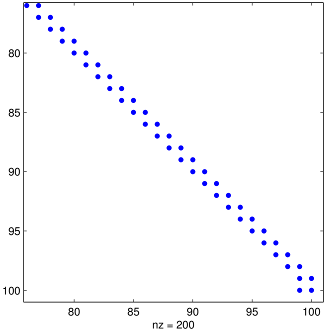


$D_{n,n+1} = \frac{1}{h}$ is ignored!

Differentiation Matrices

We can use the backward difference formula (which has the same order of accuracy) for row n instead

$$D_{n,n-1} = -\frac{1}{h}, \quad D_{nn} = \frac{1}{h}$$



Python demo: [*diff.py*] Differentiation matrices