School on spectral methods,

with applications to General Relativity and Field Theory

CIAS, Meudon Observatory, 14-18 November 2005

`http://www.lorene.obspm.fr/school/`

Lectures by

Silvano Bonazzola,
Eric Gourgoulhon,
Philippe Grandclément,
Jérôme Novak

The school has been supported by :

- Pôle numérique Relativité Meudon-Tours (Département SPM du CNRS)
- ASSNA (Action Spécifique pour la Simulation Numérique en Astrophysique)
- European Network of Theoretical Astroparticle Physics

Contents

An introduction to polynomial interpolation

Eric Gourgoulhon

Laboratoire de l'Univers et de ses Théories (LUTH)
CNRS / Observatoire de Paris
F-92195 Meudon, France

eric.gourgoulhon@obspm.fr

School on spectral methods:
Application to General Relativity and Field Theory
Meudon, 14-18 November 2005
http://www.lorene.obspm.fr/school/

Plan

Outline

Introduction

Basic idea: approximate functions $\mathbb{R} \to \mathbb{R}$ by polynomials

Polynomials are the only functions that a computer can evaluate exactely.

Two types of numerical methods based on polynomial approximations:

- **spectral methods:** high order polynomials on a single domain (or a few domains)
- **finite elements**: low order polynomials on many domains

Introduction

Basic idea: approximate functions $\mathbb{R} \to \mathbb{R}$ by polynomials

Polynomials are the only functions that a computer can evaluate exactely.

Two types of numerical methods based on polynomial approximations:

- **spectral methods**: high order polynomials on a single domain (or a few domains)
- **finite elements**: low order polynomials on many domains

Framework of this lecture

We consider real-valued functions on the compact interval $[-1,1]$:

$$f : [-1,1] \longrightarrow \mathbb{R}$$

We denote

- by $\mathbb{P}$ the set all real-valued polynomials on $[-1,1]$:

$$\forall p \in \mathbb{P},\ \forall x \in [-1,1],\ p(x) = \sum_{i=0}^{n} a_i\, x^i$$

- by $\mathbb{P}_N$ (where N is a positive integer), the subset of polynomials of degree at most N.

Is it a good idea to approximate functions by polynomials ?

For **continuous functions**, the answer is **yes**:

Theorem (Weierstrass, 1885)

$\mathbb{P}$ is a dense subspace of the space $C^0([-1,1])$ of all continuous functions on $[-1,1]$, equiped with the uniform norm $\|.\|_\infty$.[a]

[a] This is a particular case of the *Stone-Weierstrass theorem*

The uniform norm or maximum norm is defined by $\|f\|_\infty = \max_{x\in[-1,1]} |f(x)|$

Other phrasings:

For any continuous function on $[-1,1]$, f, and any $\epsilon > 0$, there exists a polynomial $p \in \mathbb{P}$ such that $\|f - p\|_\infty < \epsilon$.

For any continuous function on $[-1,1]$, f, there exists a sequence of polynomials $(p_n)_{n\in\mathbb{N}}$ which converges uniformly towards f: $\lim_{n\to\infty} \|f - p_n\|_\infty = 0$.

Best approximation polynomial

For a given continuous function: $f \in C^0([-1,1])$, a best approximation polynomial of degree N is a polynomial $p_N^*(f) \in \mathbb{P}_N$ such that

$$\|f - p_N^*(f)\|_\infty = \min\{\|f - p\|_\infty,\ p \in \mathbb{P}_N\}$$

Chebyshev's alternant theorem (or equioscillation theorem)

For any $f \in C^0([-1,1])$ and $N \geq 0$, the best approximation polynomial $p_N^*(f)$ exists and is unique. Moreover, there exists $N+2$ points x_0, x_1, ... x_{N+1} in [-1,1] such that

$$f(x_i) - p_N^*(f)(x_i) = (-1)^i \|f - p_N^*(f)\|_\infty, \qquad 0 \leq i \leq N+1$$
$$\text{or } f(x_i) - p_N^*(f)(x_i) = (-1)^{i+1} \|f - p_N^*(f)\|_\infty, \qquad 0 \leq i \leq N+1$$

Corollary: $p_N^*(f)$ interpolates f in $N+1$ points.

Illustration of Chebyshev's alternant theorem

$N = 1$

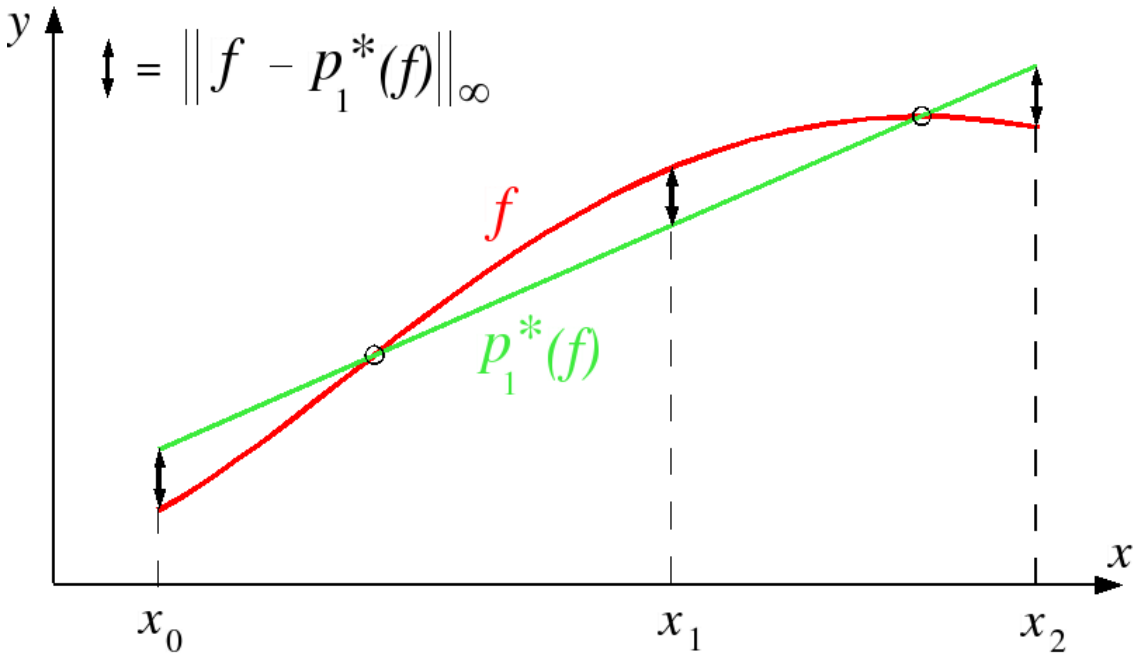
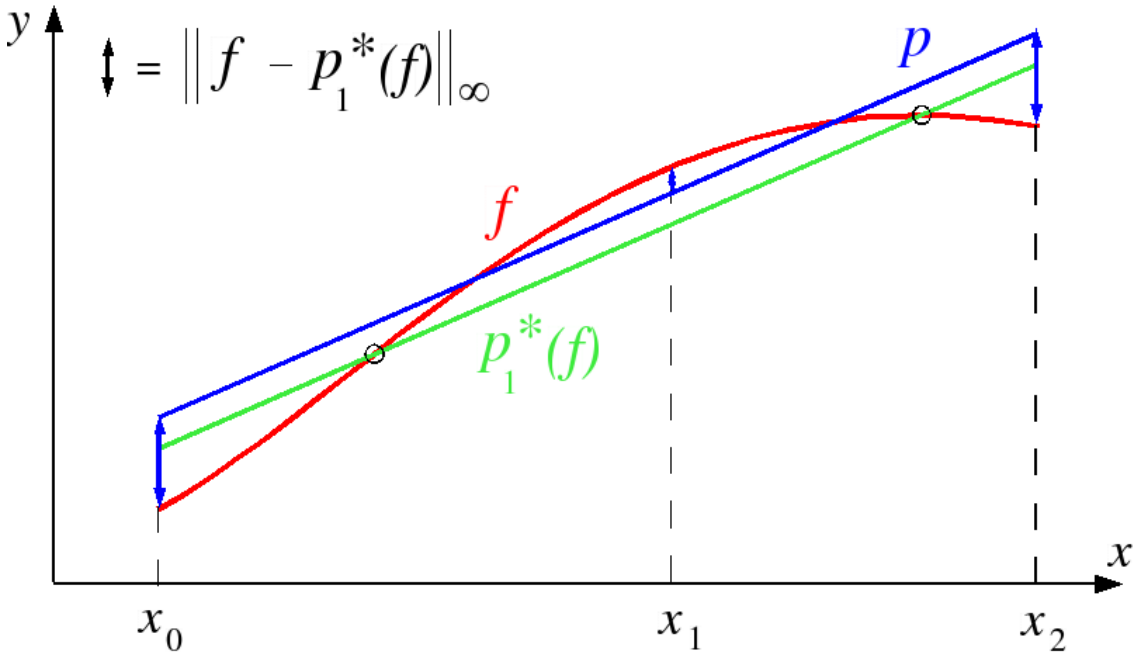


Illustration of Chebyshev's alternant theorem

$N = 1$



Outline

Interpolation on an arbitrary grid

Definition: given an integer $N \geq 1$, a grid is a set of $N+1$ points $X = (x_i)_{0 \leq i \leq N}$ in [-1,1] such that $-1 \leq x_0 < x_1 < \cdots < x_N \leq 1$. The $N+1$ points $(x_i)_{0 \leq i \leq N}$ are called the nodes of the grid.

Theorem

Given a function $f \in C^0([-1,1])$ and a grid of $N+1$ nodes, $X = (x_i)_{0 \leq i \leq N}$, there exist a unique polynomial of degree N, $I_N^X f$, such that

$$I_N^X f(x_i) = f(x_i), \qquad 0 \leq i \leq N$$

$I_N^X f$ is called the interpolant (or the interpolating polynomial) of f through the grid X.

Lagrange form of the interpolant

The interpolant $I_N^X f$ can be expressed in the *Lagrange form:*

$$I_N^X f(x) = \sum_{i=0}^{N} f(x_i)\, \ell_i^X(x),$$

where $\ell_i^X(x)$ is the i-th Lagrange cardinal polynomial associated with the grid X:

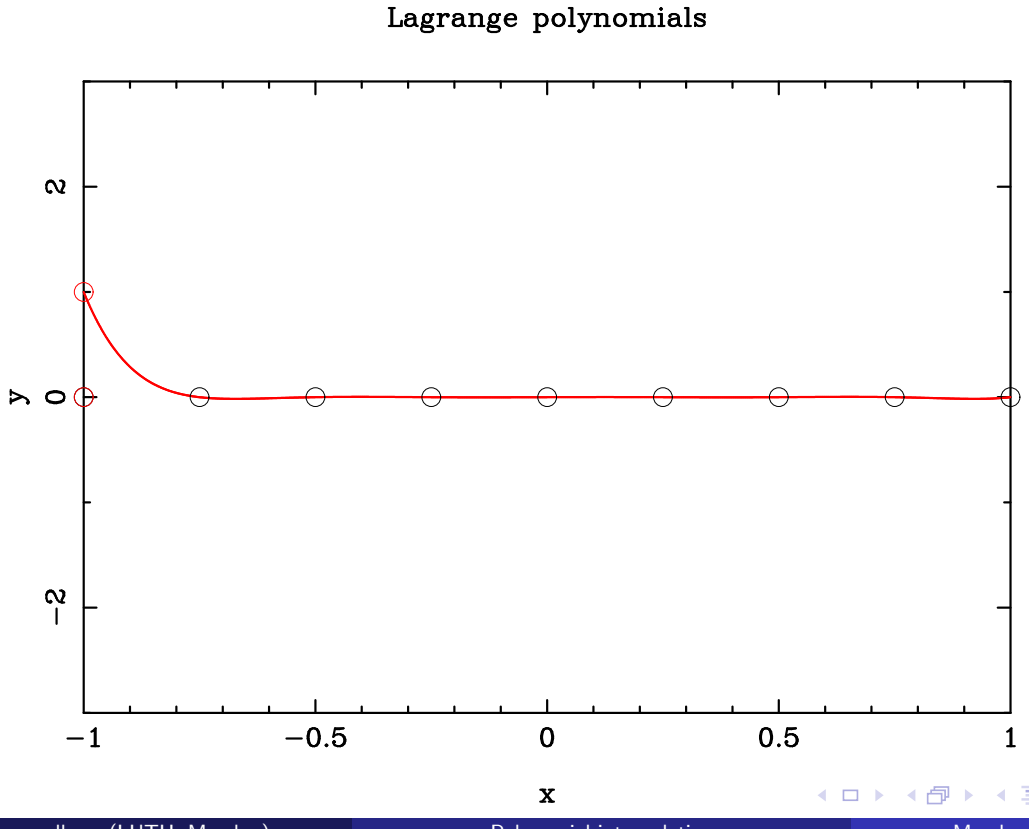
$$\ell_i^X(x) := \prod_{\substack{j=0 \\ j \neq i}}^{N} \frac{x - x_j}{x_i - x_j}, \qquad 0 \leq i \leq N$$

The Lagrange cardinal polynomials are such that

$$\ell_i^X(x_j) = \delta_{ij}, \qquad 0 \leq i, j \leq N$$

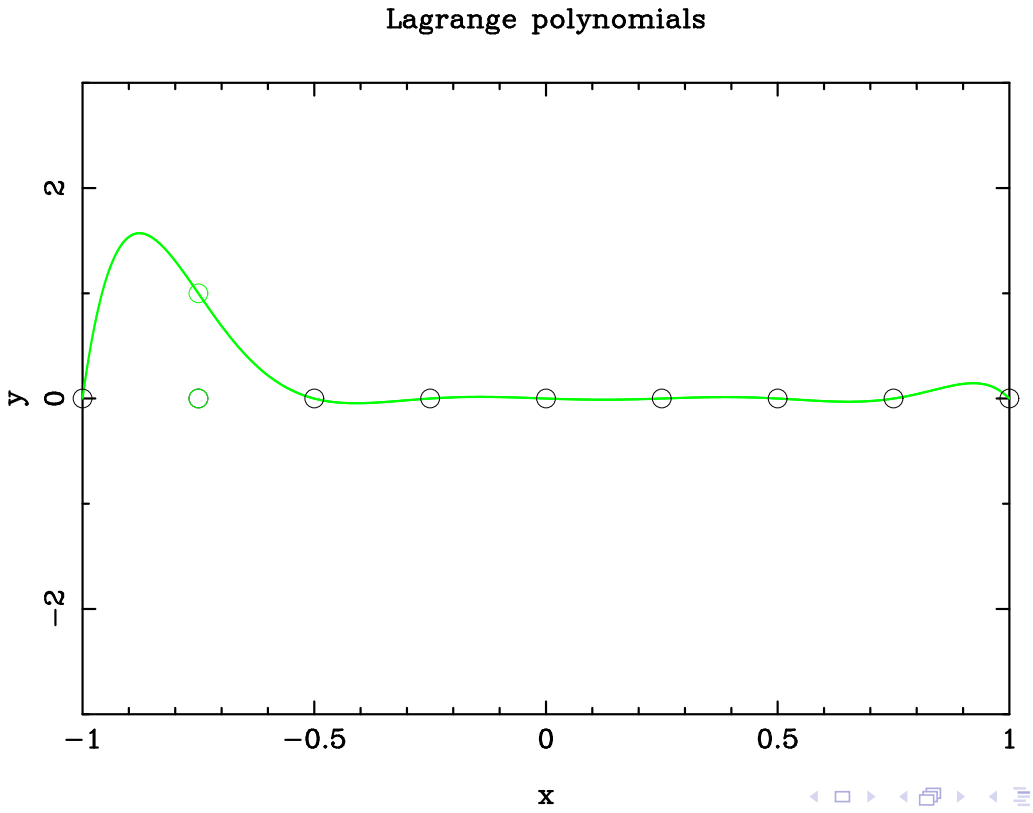
Examples of Lagrange polynomials

Uniform grid $N = 8$ $\qquad \ell_0^X(x)$



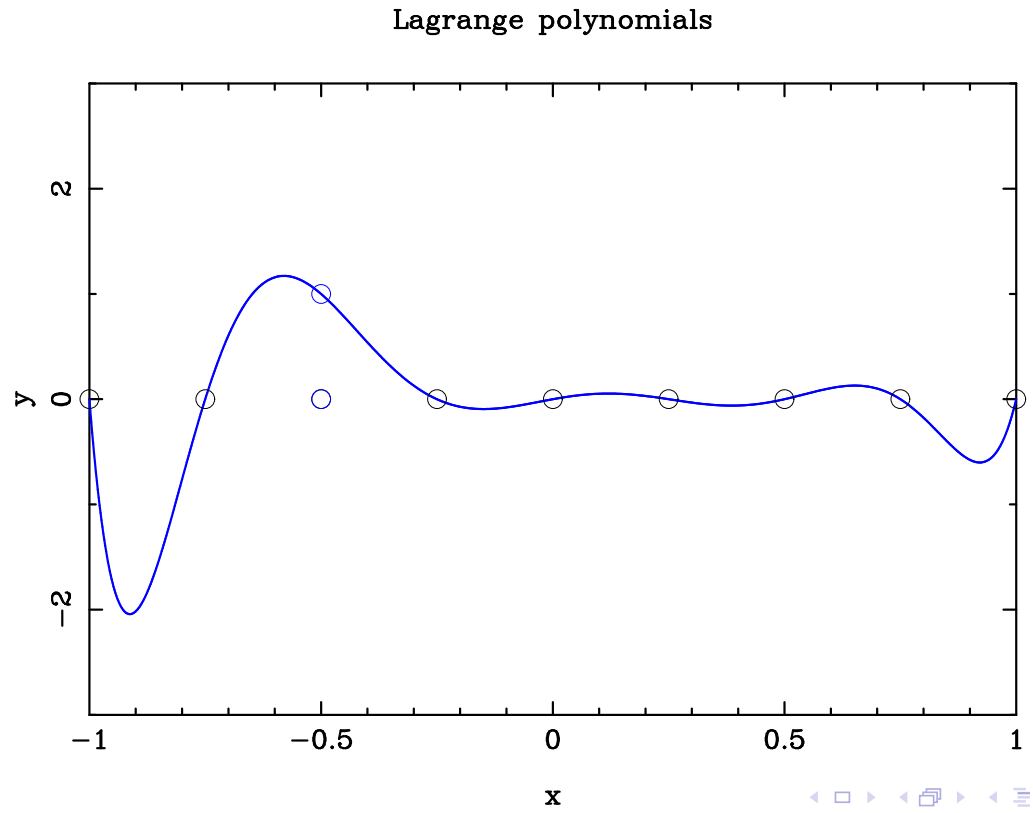
Examples of Lagrange polynomials

Uniform grid $N = 8$ $\quad \ell_1^X(x)$



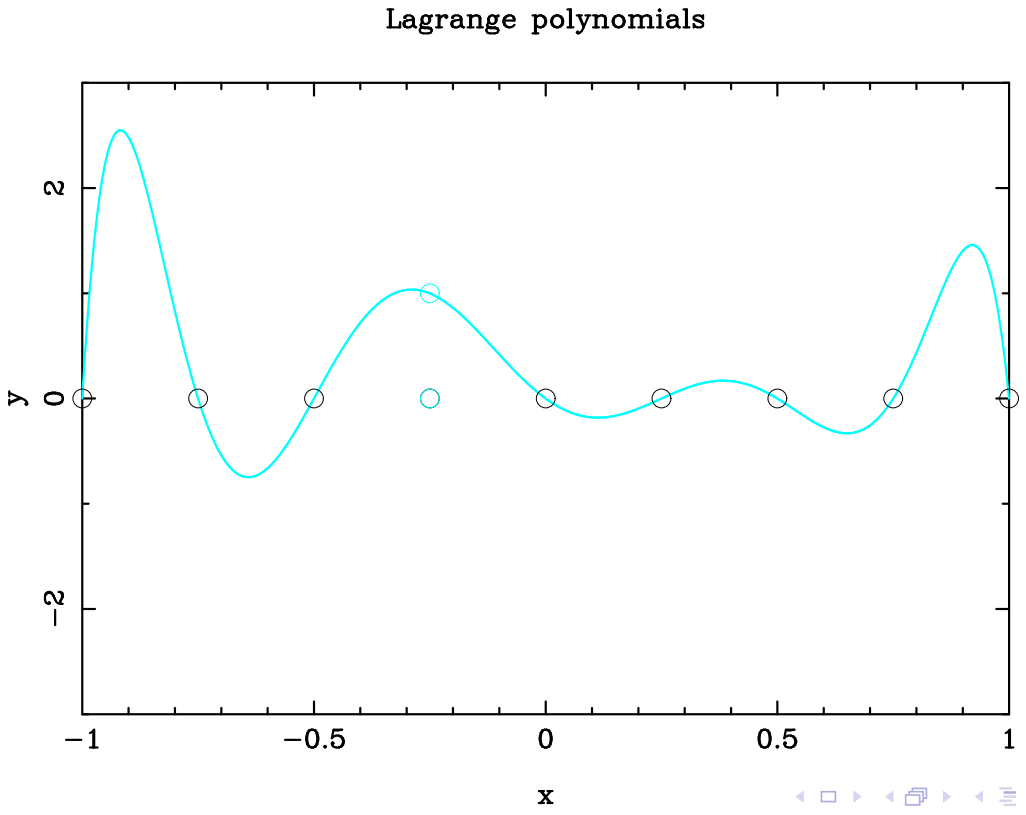
Examples of Lagrange polynomials

Uniform grid $N = 8$ $\quad \ell_2^X(x)$



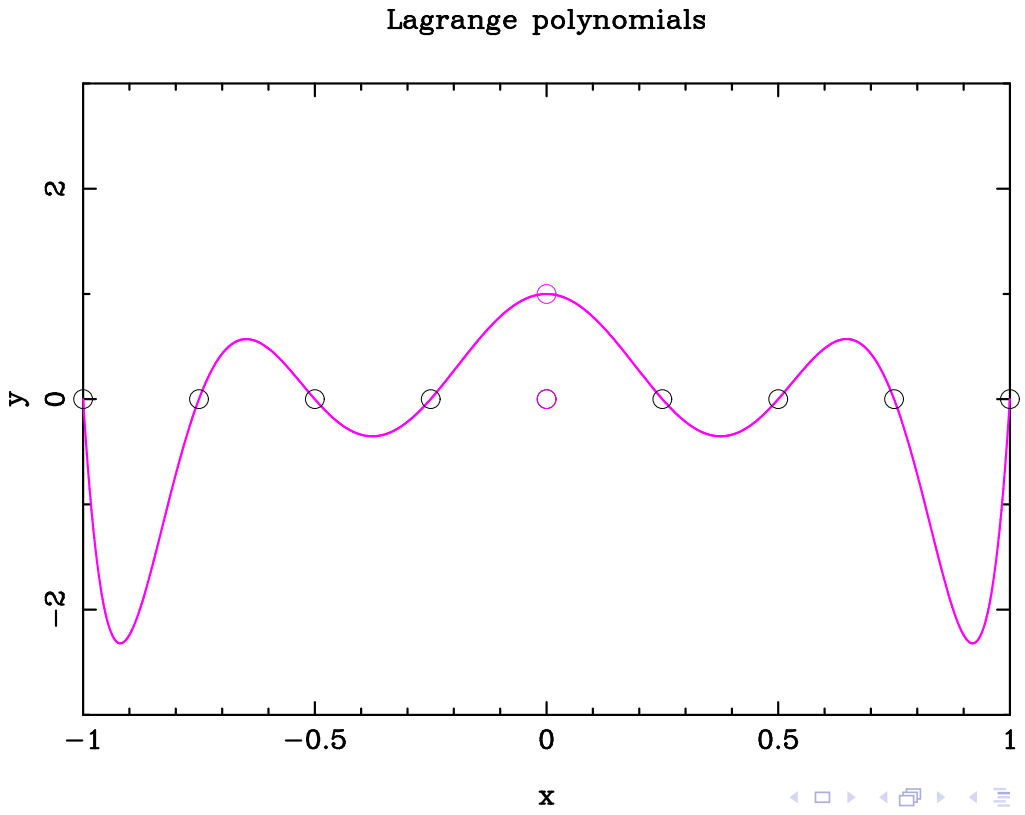
Examples of Lagrange polynomials

Uniform grid $N = 8$ $\quad \ell_3^X(x)$



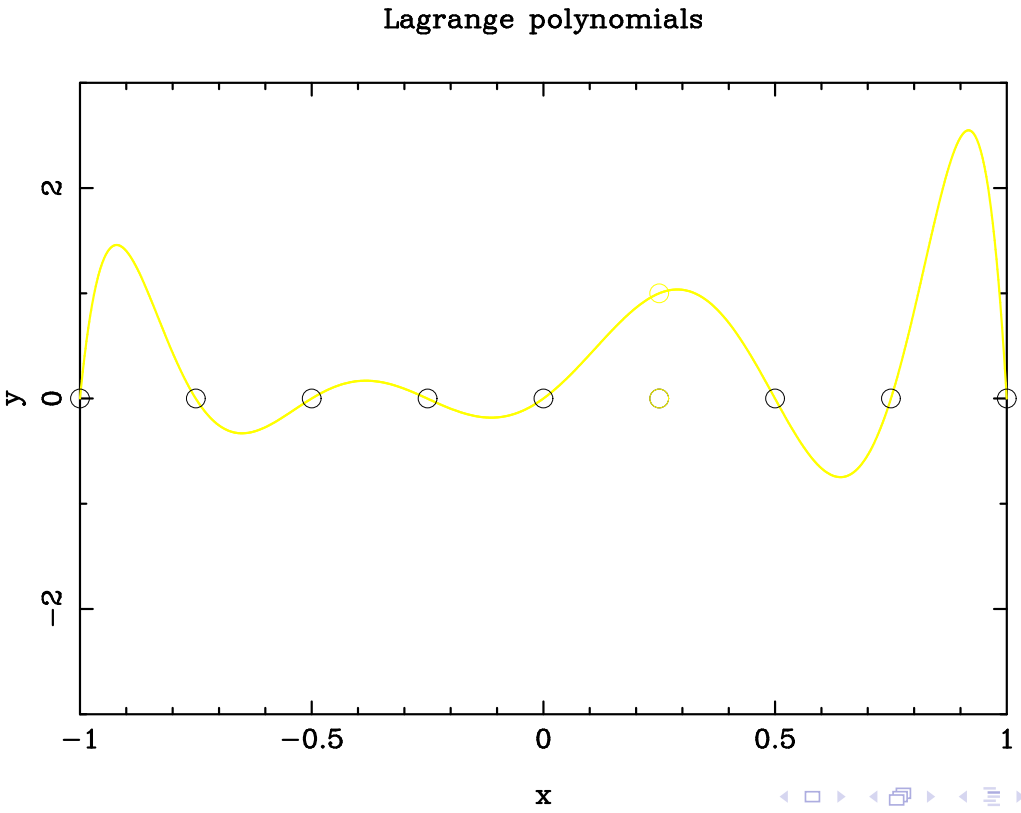
Examples of Lagrange polynomials

Uniform grid $N = 8$ $\quad \ell_4^X(x)$



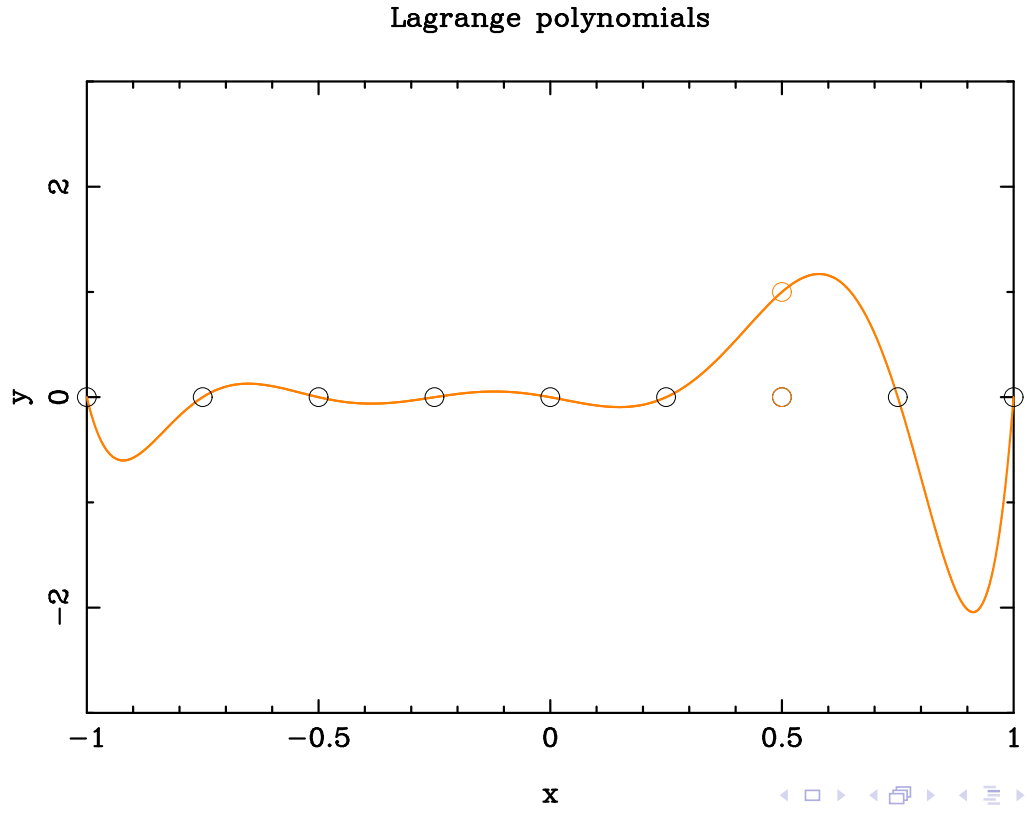
Examples of Lagrange polynomials

Uniform grid $N = 8$ $\quad \ell_5^X(x)$



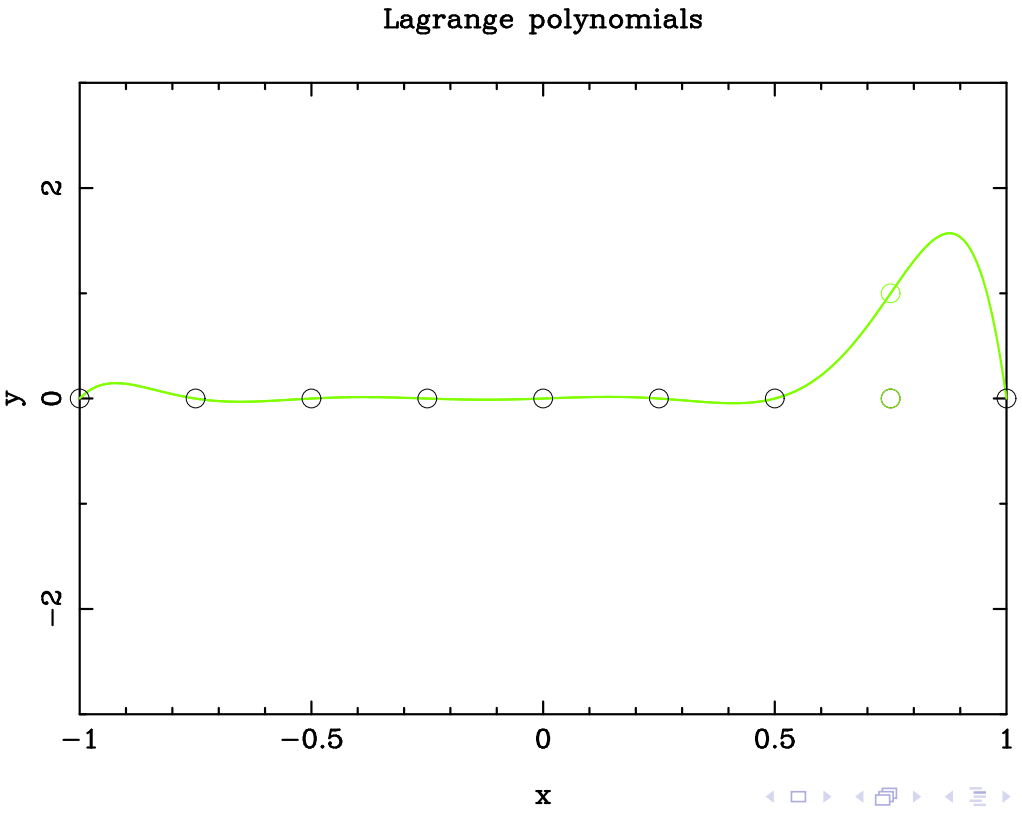
Examples of Lagrange polynomials

Uniform grid $N = 8$ $\quad \ell_6^X(x)$



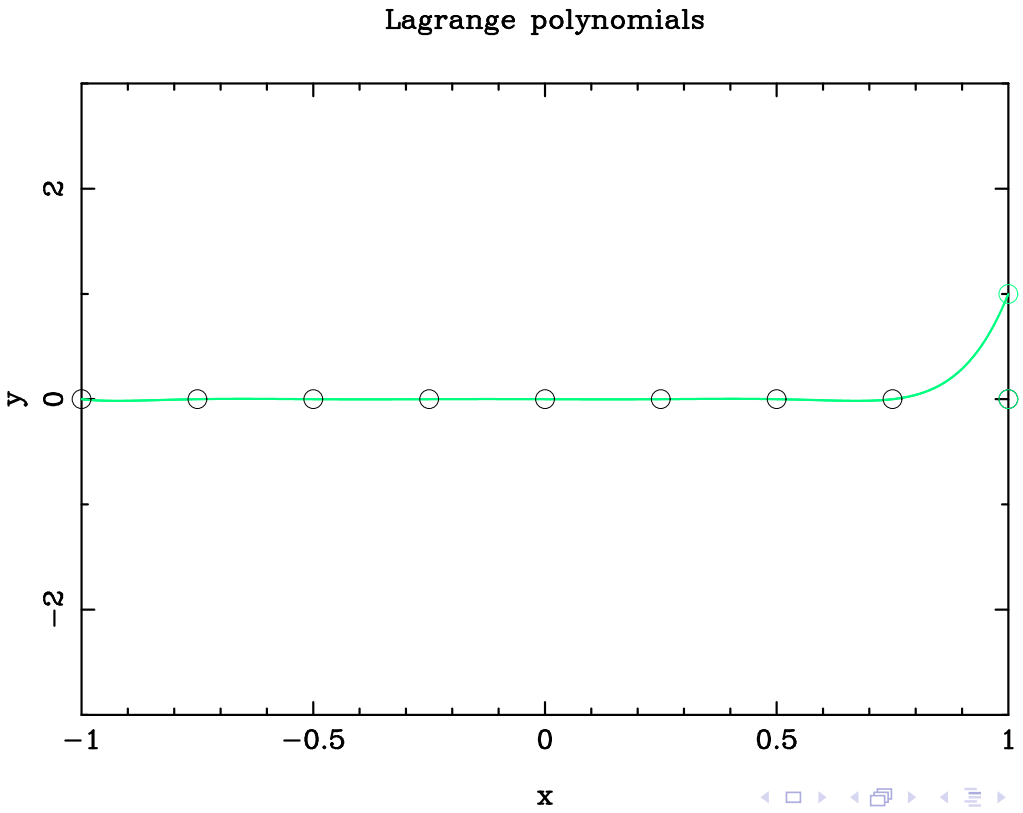
Examples of Lagrange polynomials

Uniform grid $N = 8$ $\quad \ell_7^X(x)$



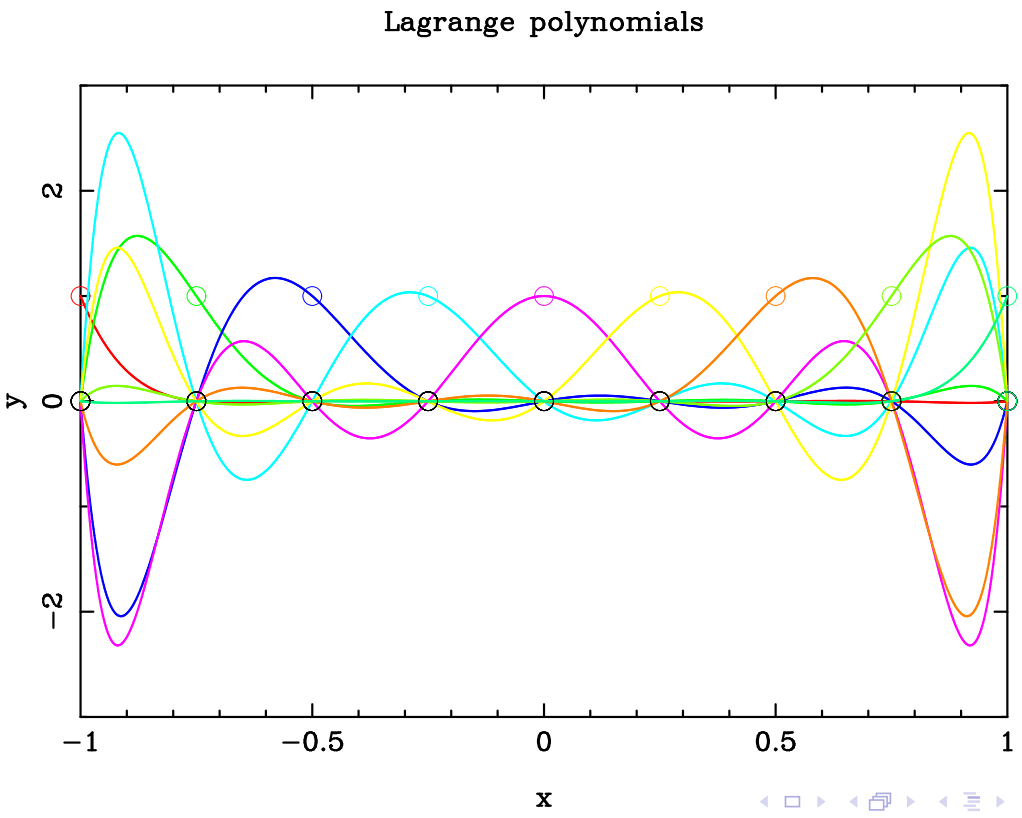
Examples of Lagrange polynomials

Uniform grid $N = 8$ $\quad \ell_8^X(x)$



Examples of Lagrange polynomials

Uniform grid $N = 8$



Interpolation error with respect to the best approximation error

Let $N \in \mathbb{N}$, $X = (x_i)_{0 \leq i \leq N}$ a grid of $N+1$ nodes and $f \in C^0([-1,1])$.

Let us consider the interpolant $I_N^X f$ of f through the grid X.

The best approximation polynomial $p_N^*(f)$ is also an interpolant of f at $N+1$ nodes (in general different from X) reminder

How does the error $\|f - I_N^X f\|_\infty$ behave with respect to the smallest possible error $\|f - p_N^*(f)\|_\infty$?

The answer is given by the formula:

$$\|f - I_N^X f\|_\infty \leq (1 + \Lambda_N(X)) \, \|f - p_N^*(f)\|_\infty$$

where $\Lambda_N(X)$ is the Lebesgue constant relative to the grid X:

$$\Lambda_N(X) := \max_{x \in [-1,1]} \sum_{i=0}^{N} \left| \ell_i^X(x) \right|$$

Lebesgue constant

The Lebesgue constant contains all the information on the effects of the choice of X on $\|f - I_N^X f\|_\infty$.

Theorem (Erdős, 1961)

For any choice of the grid X, there exists a constant $C > 0$ such that

$$\Lambda_N(X) > \frac{2}{\pi} \ln(N+1) - C$$

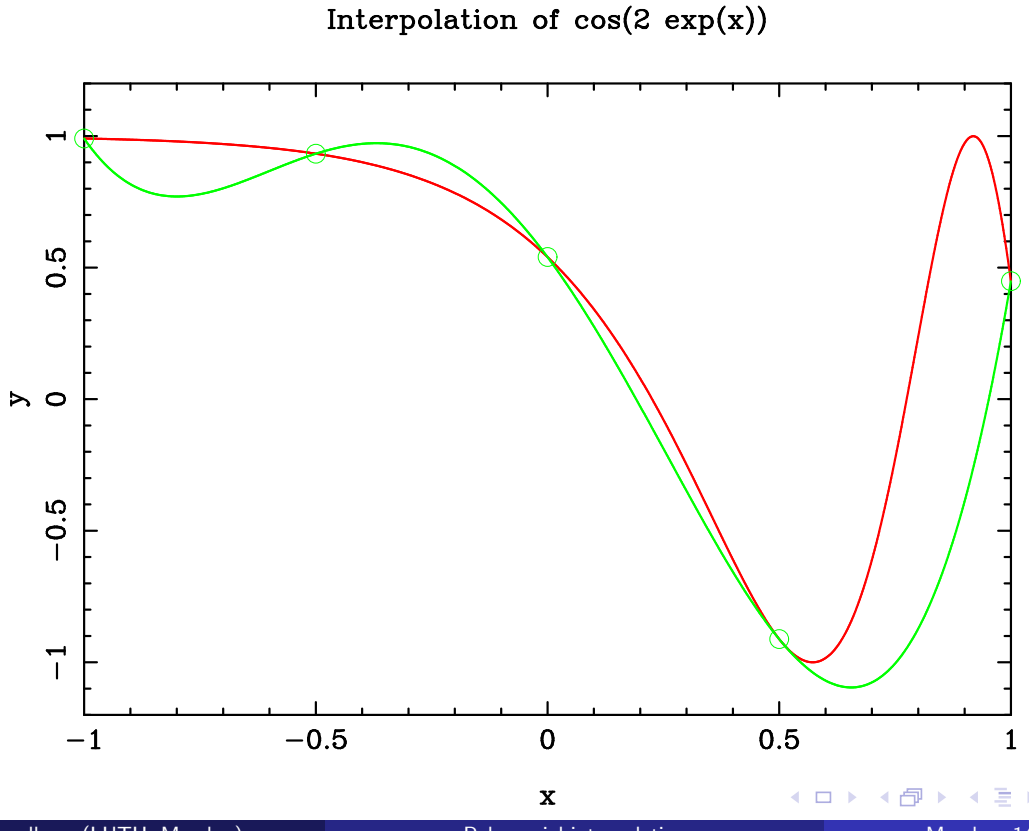
Corollary: $\Lambda_N(X) \to \infty$ as $N \to \infty$

In particular, for a **uniform** grid, $\Lambda_N(X) \sim \dfrac{2^{N+1}}{eN \ln N}$ as $N \to \infty$!

This means that for any choice of type of sampling of $[-1, 1]$, there exists a continuous function $f \in C^0([-1, 1])$ such that $I_N^X f$ does not convergence uniformly towards f.

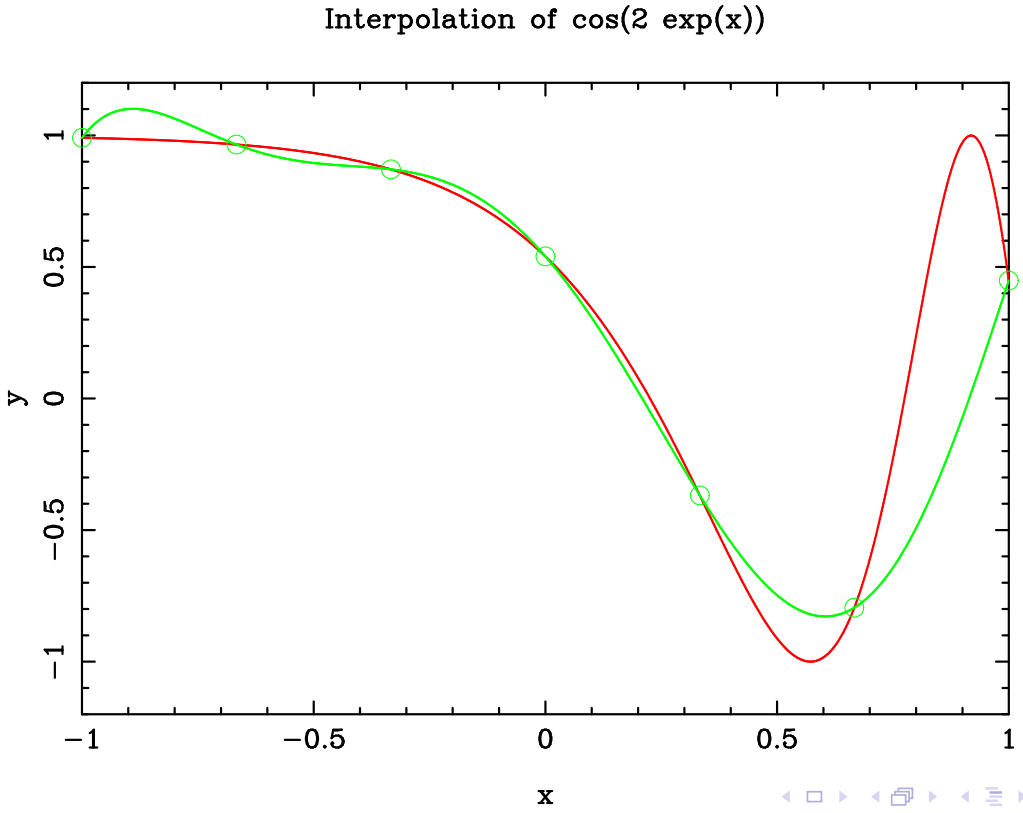
Example: uniform interpolation of a "gentle" function

$f(x) = \cos(2\exp(x))$ uniform grid $N = 4$: $\|f - I_4^X f\|_\infty \simeq 1.40$



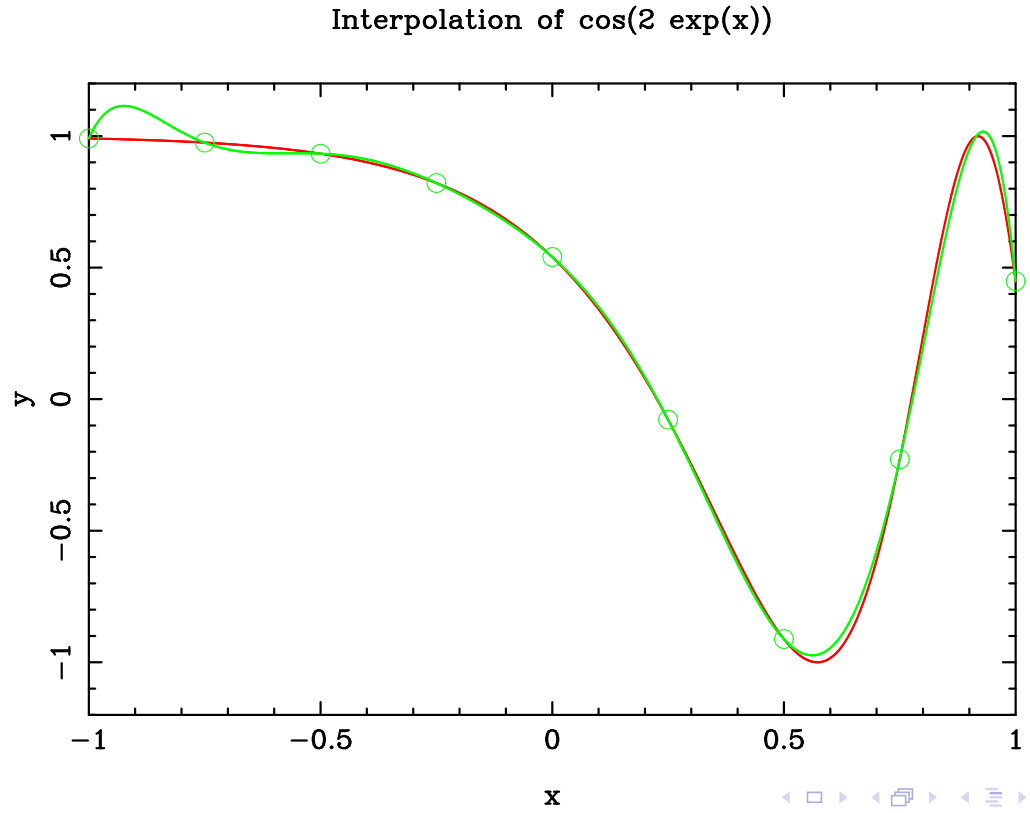
Example: uniform interpolation of a "gentle" function

$f(x) = \cos(2\exp(x))$ uniform grid $N = 6$: $\|f - I_6^X f\|_\infty \simeq 1.05$



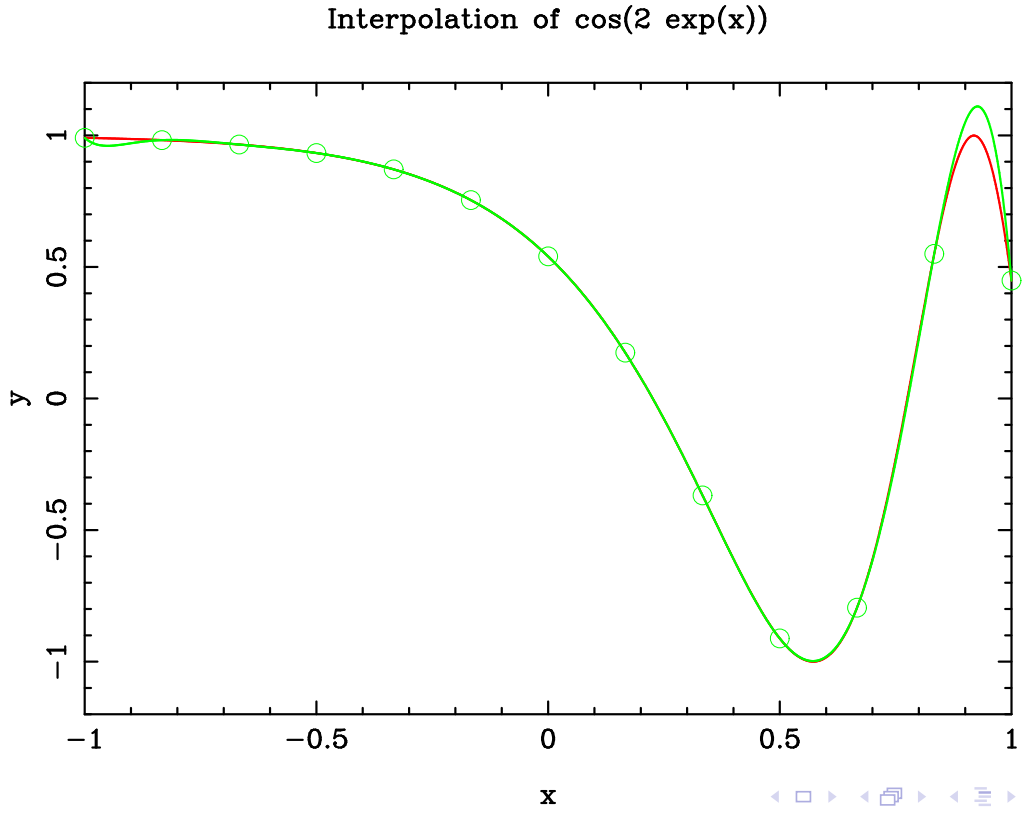
Example: uniform interpolation of a "gentle" function

$f(x) = \cos(2\exp(x))$ uniform grid $N = 8$: $\|f - I_8^X f\|_\infty \simeq 0.13$



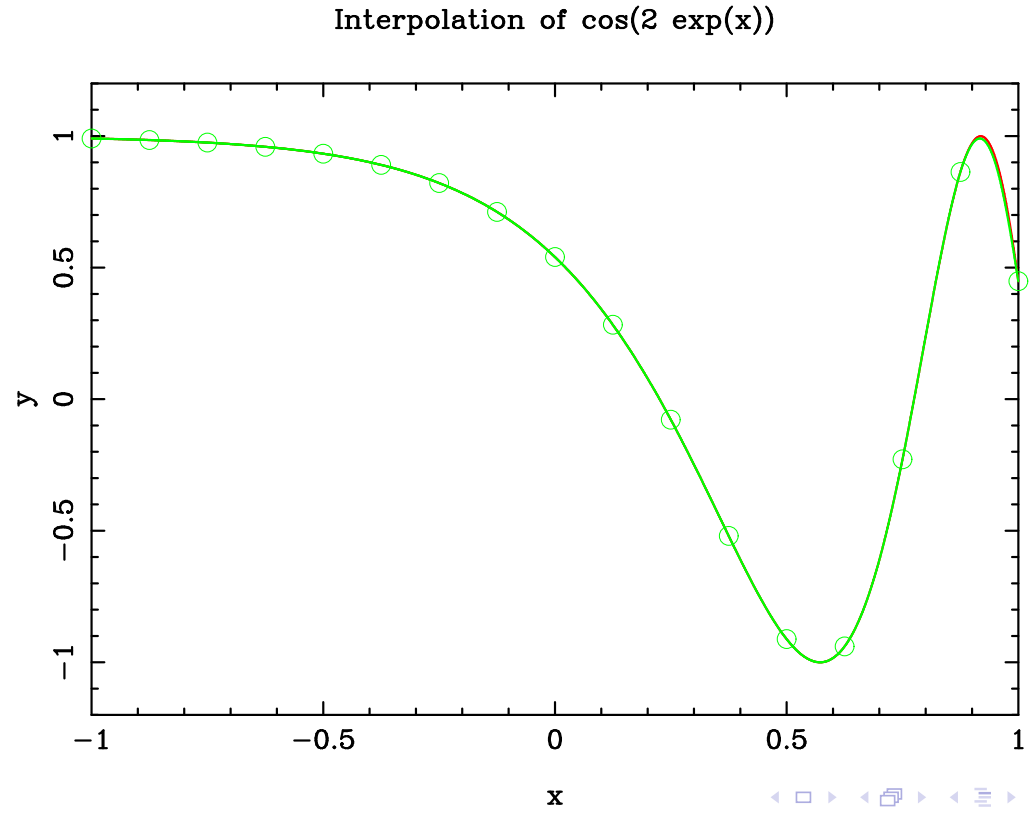
Example: uniform interpolation of a "gentle" function

$f(x) = \cos(2\exp(x))$ uniform grid $N = 12$: $\|f - I_{12}^X f\|_\infty \simeq 0.13$



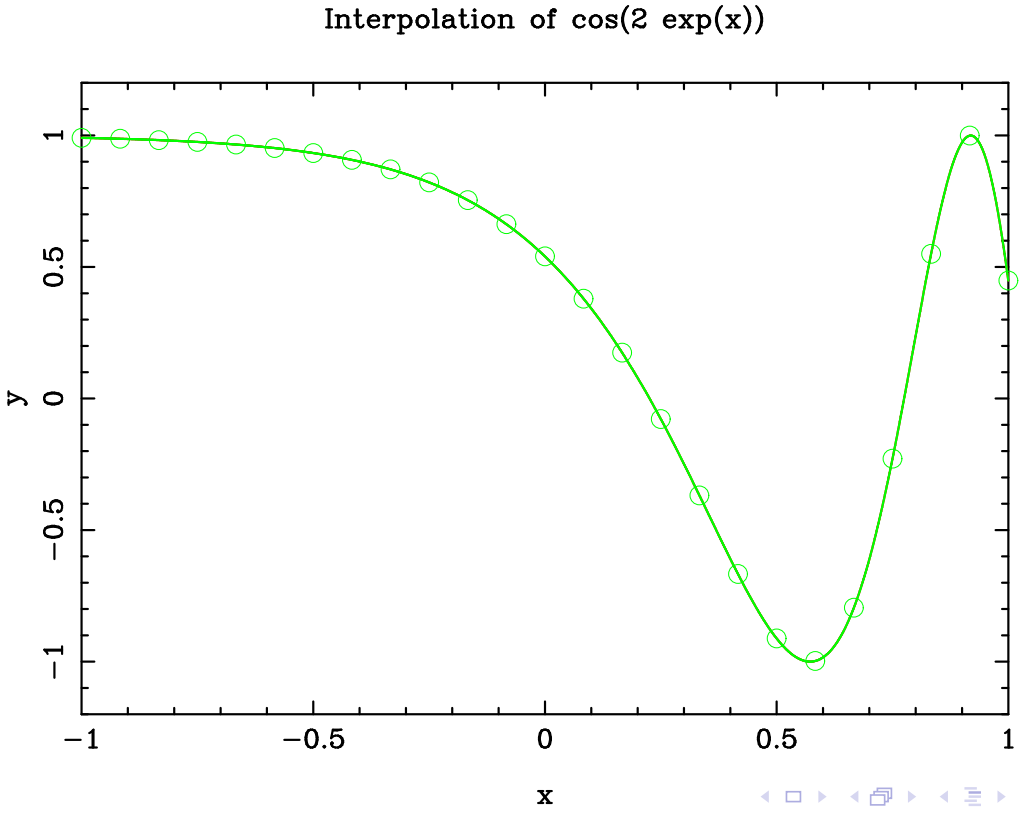
Example: uniform interpolation of a "gentle" function

$f(x) = \cos(2\exp(x))$ uniform grid $N = 16$: $\|f - I_{16}^X f\|_\infty \simeq 0.025$



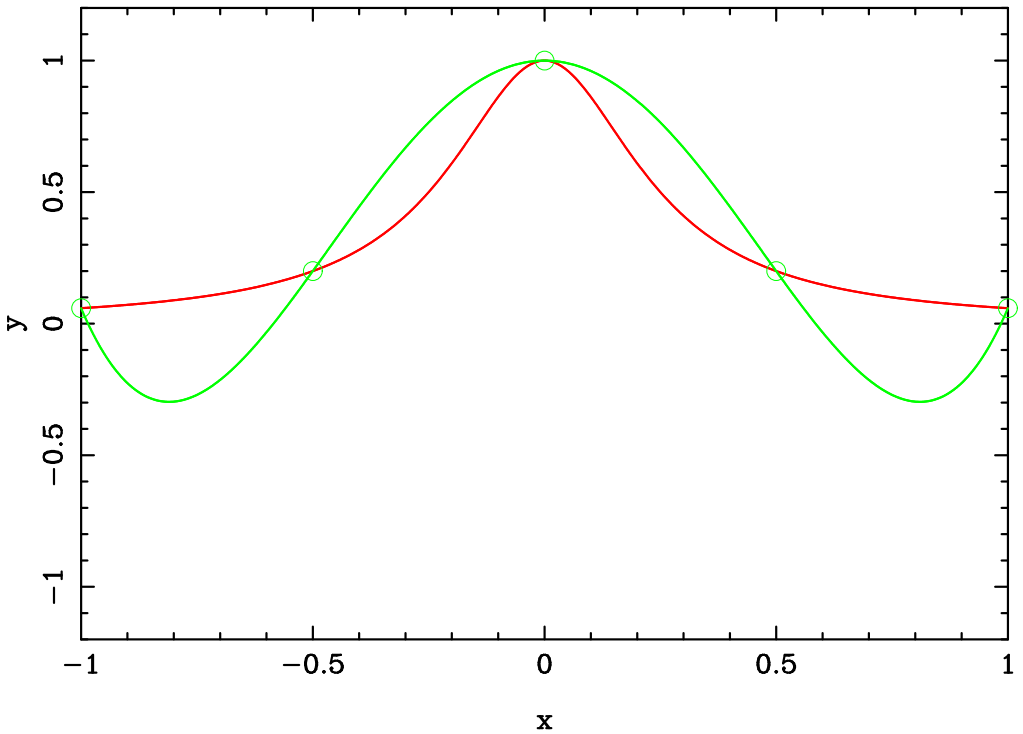
Example: uniform interpolation of a "gentle" function

$f(x) = \cos(2\exp(x))$ uniform grid $N = 24$: $\|f - I_{24}^X f\|_\infty \simeq 4.6\,10^{-4}$



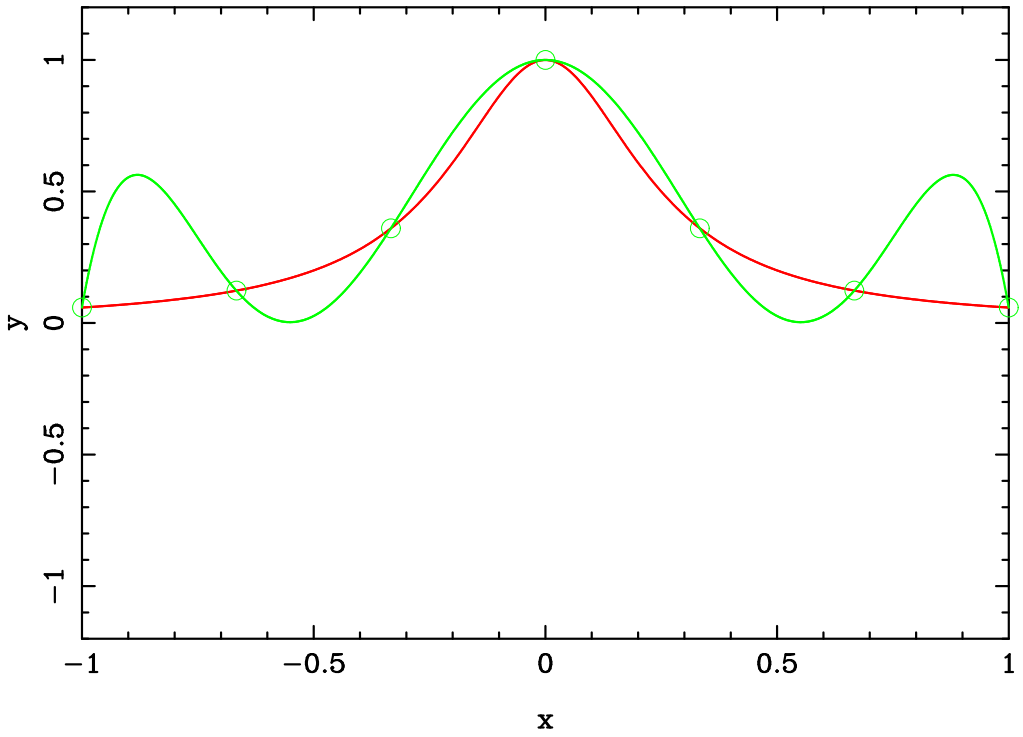
Runge phenomenon

$f(x) = \dfrac{1}{1+16x^2}$ uniform grid $N = 4$: $\|f - I_4^X f\|_\infty \simeq 0.39$



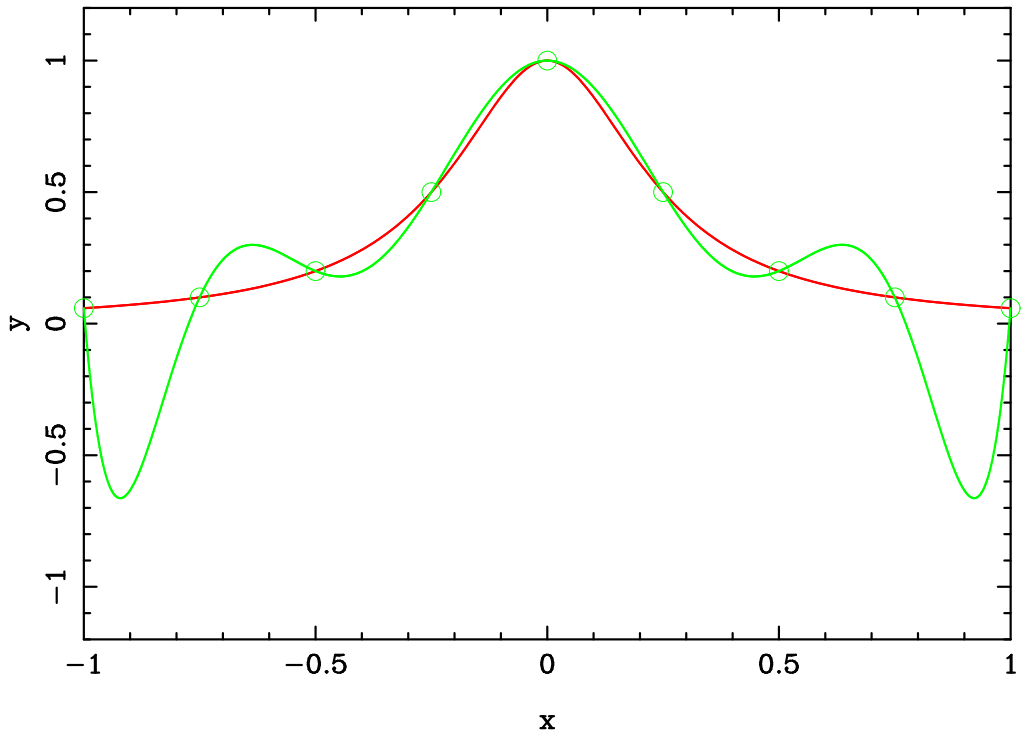
Runge phenomenon

$$f(x) = \frac{1}{1+16x^2} \qquad \text{uniform grid } N = 6 : \ \|f - I_6^X f\|_\infty \simeq 0.49$$



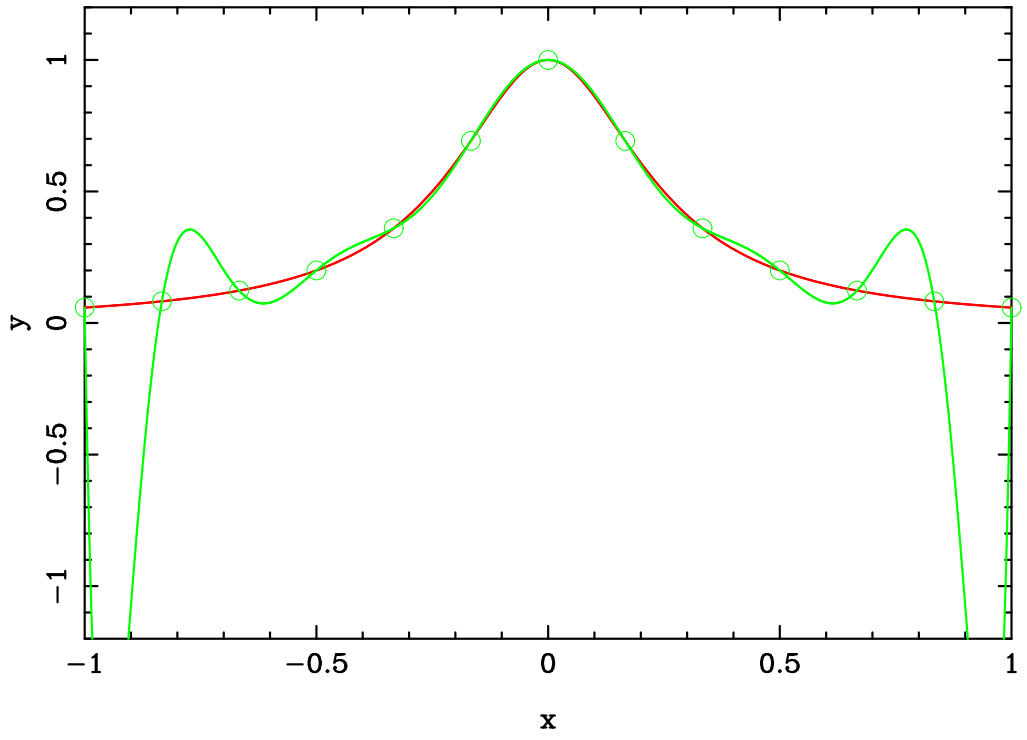
Runge phenomenon

$$f(x) = \frac{1}{1+16x^2} \qquad \text{uniform grid } N = 8 : \ \|f - I_8^X f\|_\infty \simeq 0.73$$



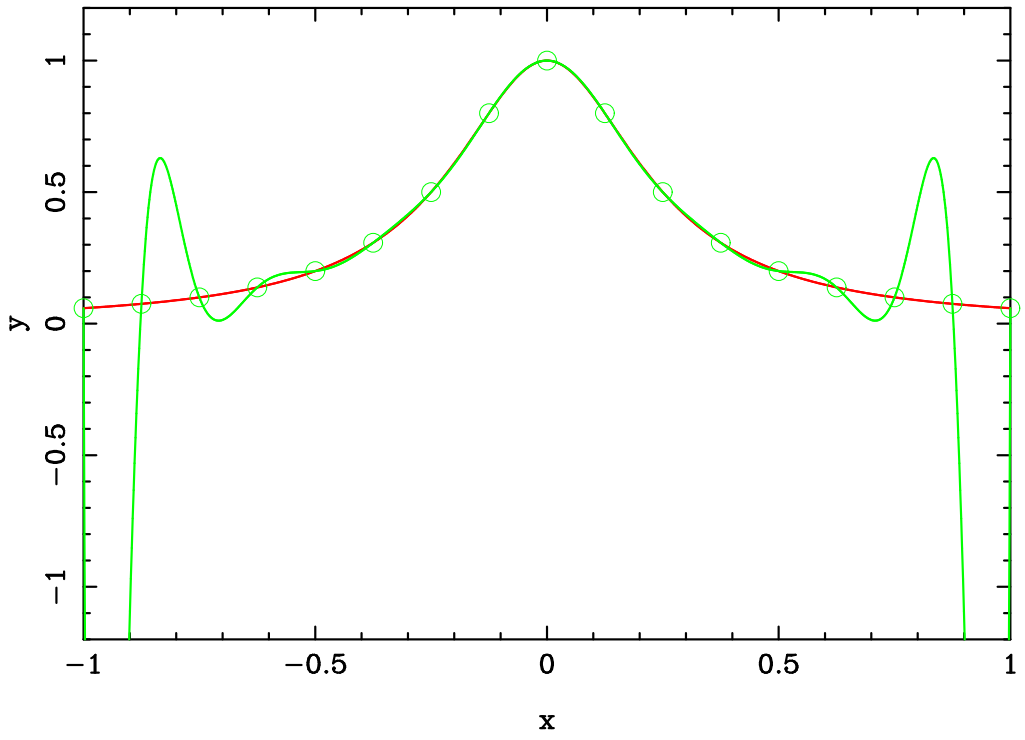
Runge phenomenon

$f(x) = \dfrac{1}{1+16x^2}$ uniform grid $N = 12$: $\|f - I_{12}^X f\|_\infty \simeq 1.97$



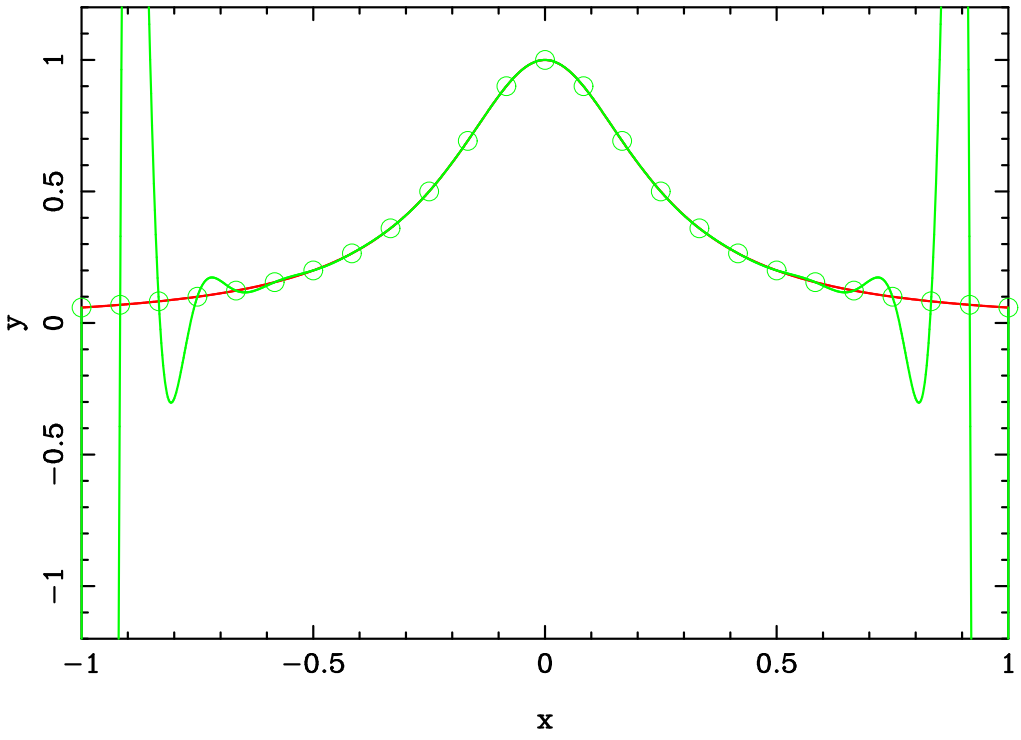
Runge phenomenon

$f(x) = \dfrac{1}{1+16x^2}$ uniform grid $N = 16$: $\|f - I_{16}^X f\|_\infty \simeq 5.9$



Runge phenomenon

$$f(x) = \frac{1}{1+16x^2} \qquad \text{uniform grid } N = 24 : \ \|f - I_{24}^X f\|_\infty \simeq 62$$



Evaluation of the interpolation error

Let us assume that the function f is sufficiently smooth to have derivatives at least up to the order $N+1$, with $f^{(N+1)}$ continuous, i.e. $f \in C^{N+1}([-1,1])$.

Theorem (Cauchy)

If $f \in C^{N+1}([-1,1])$, then for any grid X of $N+1$ nodes, and for any $x \in [-1,1]$, the interpolation error at x is

$$f(x) - I_N^X(x) = \frac{f^{(N+1)}(\xi)}{(N+1)!}\, \omega_{N+1}^X(x) \tag{1}$$

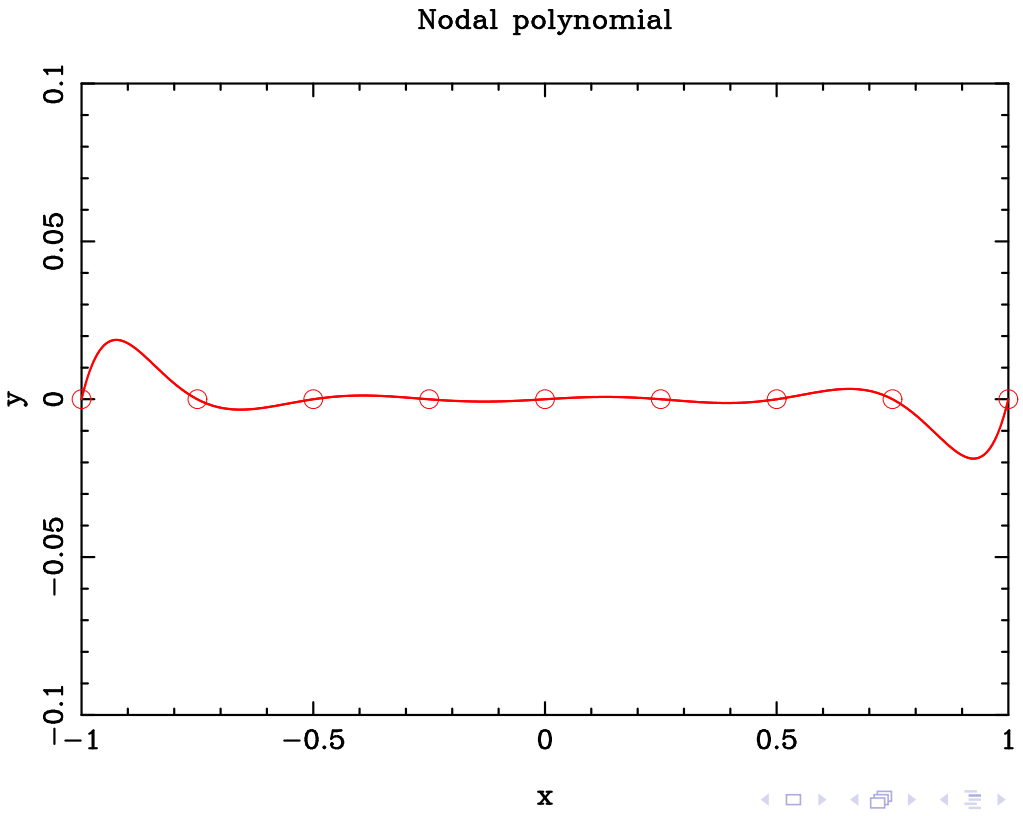
where $\xi = \xi(x) \in [-1,1]$ and $\omega_{N+1}^X(x)$ is the nodal polynomial associated with the grid X.

Definition: The nodal polynomial associated with the grid X is the unique polynomial of degree $N+1$ and leading coefficient 1 whose zeros are the $N+1$ nodes of X:

$$\boxed{\omega_{N+1}^X(x) := \prod_{i=0}^{N} (x - x_i)}$$

Example of nodal polynomial

Uniform grid $N = 8$



Minimizing the interpolation error by the choice of grid

In Eq. (1), we have no control on $f^{(N+1)}$, which can be large.
For example, for $f(x) = 1/(1+\alpha^2 x^2)$, $\|f^{(N+1)}\|_\infty = (N+1)!\,\alpha^{N+1}$.

Idea: choose the grid X so that $\omega_{N+1}^X(x)$ is small, i.e. $\|\omega_{N+1}^X\|_\infty$ is small.

Notice: $\omega_{N+1}^X(x)$ has leading coefficient 1: $\omega_{N+1}^X(x) = x^{N+1} + \sum_{i=0}^{N} a_i\, x^i$.

Theorem (Chebyshev)

Among all the polynomials of degree $N+1$ and leading coefficient 1, the unique polynomial which has the smallest uniform norm on $[-1,1]$ is the $(N+1)$-th **Chebyshev polynomial** *divided by 2^N: $T_{N+1}(x)/2^N$.*

Since $\|T_{N+1}\|_\infty = 1$, we conclude that if we choose the grid nodes $(x_i)_{0\leq i\leq N}$ to be the $N+1$ zeros of the Chebyshev polynomial T_{N+1}, we have

$$\|\omega_{N+1}^X\|_\infty = \frac{1}{2^N}$$

and this is the smallest possible value.

Chebyshev-Gauss grid

The grid $X = (x_i)_{0 \leq i \leq N}$ such that the x_i's are the $N+1$ zeros of the Chebyshev polynomial of degree $N+1$ is called the Chebyshev-Gauss (CG) grid.
It has much better interpolation properties than the uniform grid considered so far.
In particular, from Eq. (1), for any function $f \in C^{N+1}([-1,1])$,

$$\left\| f - I_N^{\mathrm{CG}} f \right\|_\infty \leq \frac{1}{2^N (N+1)!} \left\| f^{(N+1)} \right\|_\infty$$

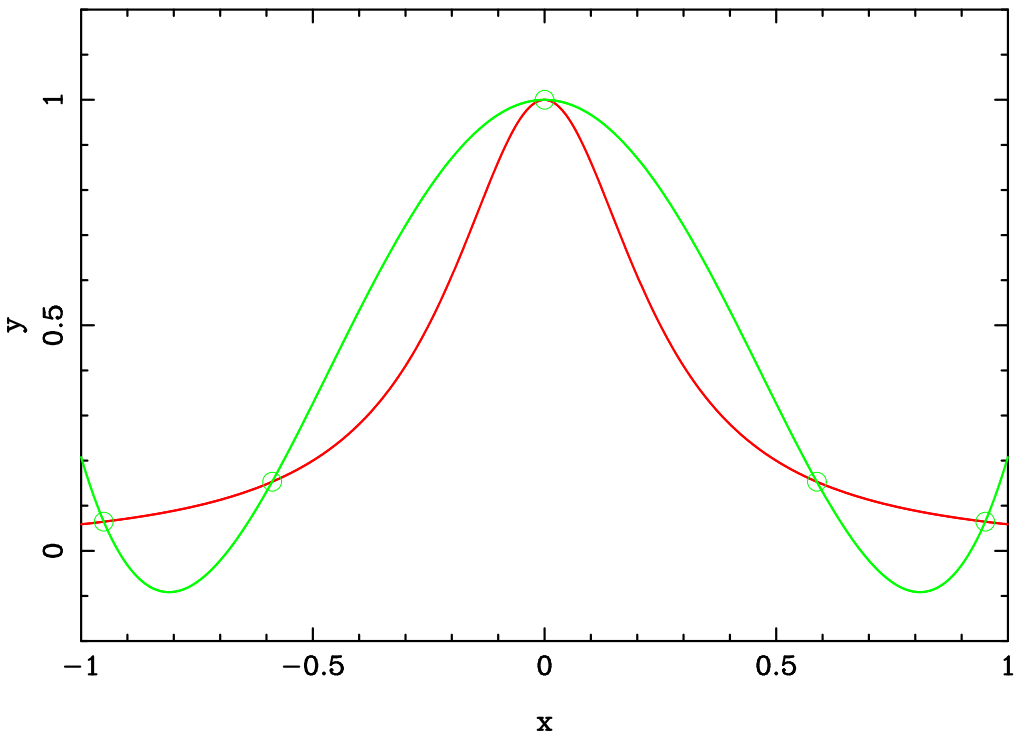
If $f^{(N+1)}$ is uniformly bounded, the convergence of the interpolant $I_N^{\mathrm{CG}} f$ towards f when $N \to \infty$ is then extremely fast.
Also the Lebesgue constant associated with the Chebyshev-Gauss grid is small:

$$\Lambda_N(CG) \sim \frac{2}{\pi} \ln(N+1) \quad \text{as} \quad N \to \infty$$

This is much better than uniform grids and close to the optimal value reminder

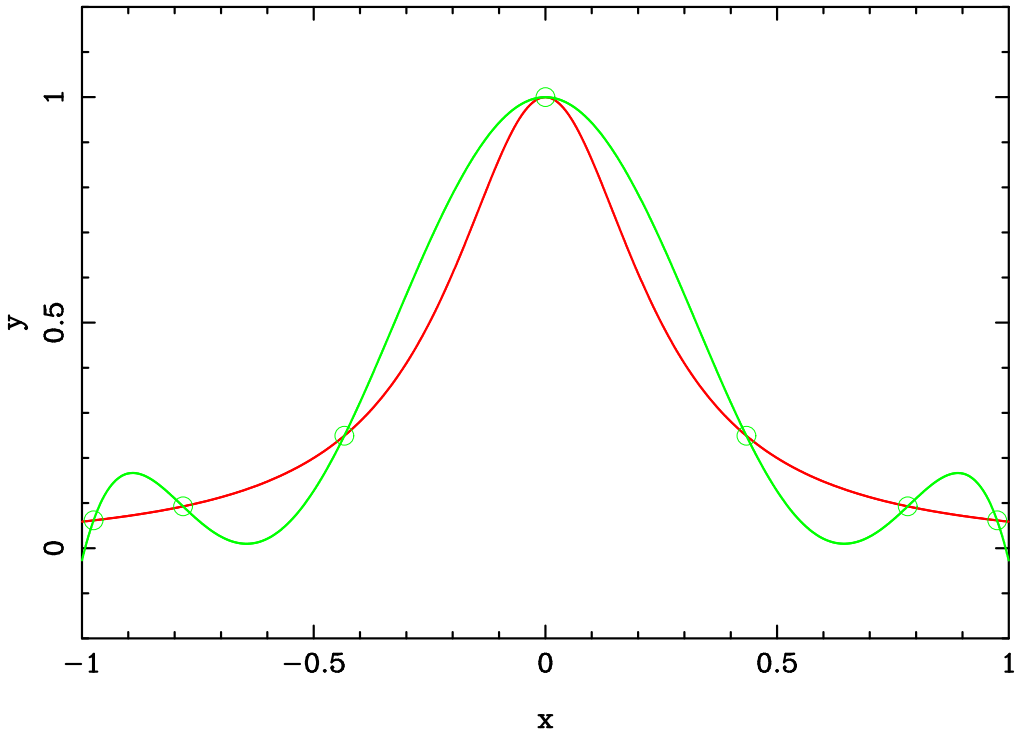
Example: Chebyshev-Gauss interpolation of $f(x) = \frac{1}{1+16x^2}$

$f(x) = \dfrac{1}{1+16x^2}$ CG grid $N = 4$: $\|f - I_4^{\mathrm{CG}} f\|_\infty \simeq 0.31$



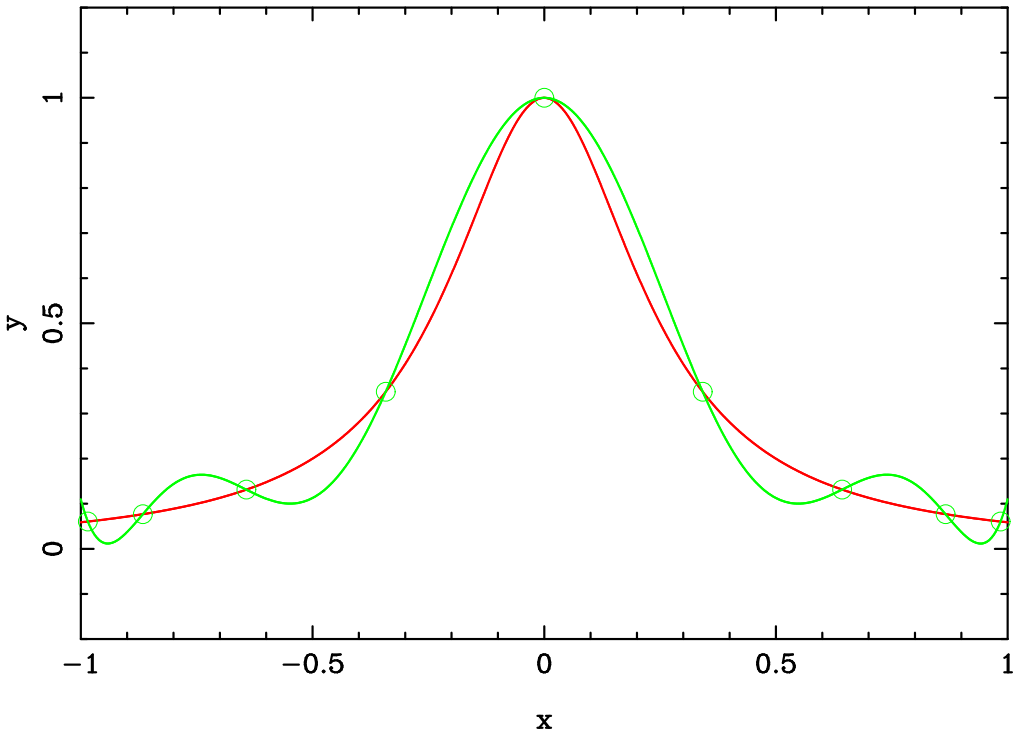
Example: Chebyshev-Gauss interpolation of $f(x) = \frac{1}{1+16x^2}$

$f(x) = \dfrac{1}{1+16x^2}$ CG grid $N = 6$: $\|f - I_6^{CG} f\|_\infty \simeq 0.18$



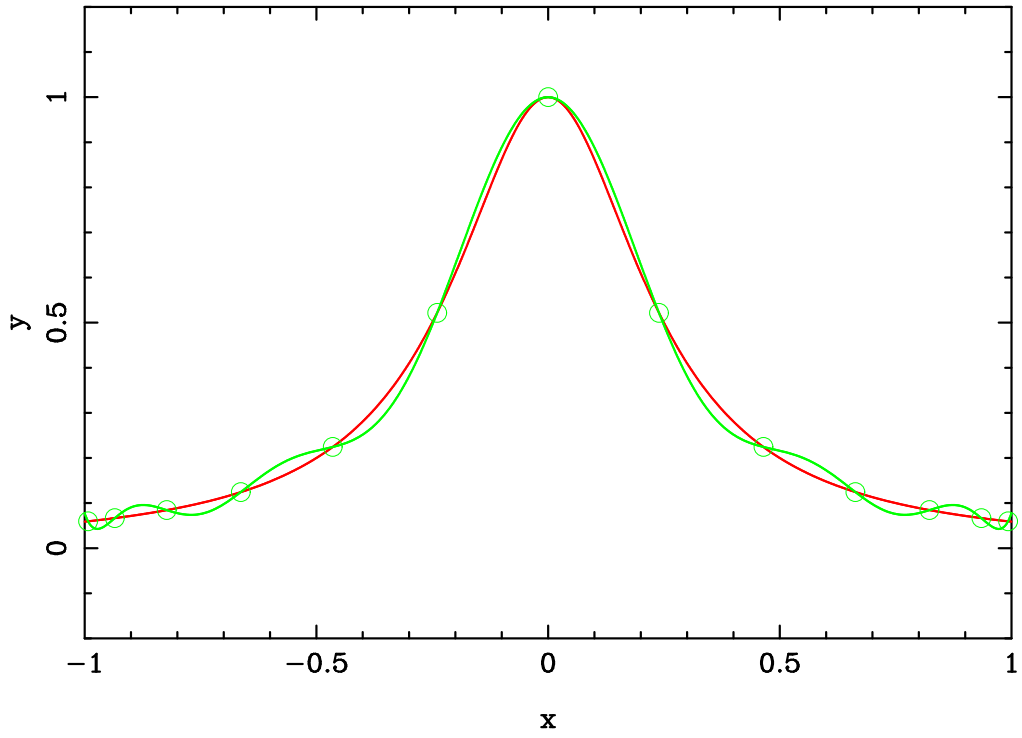
Example: Chebyshev-Gauss interpolation of $f(x) = \frac{1}{1+16x^2}$

$f(x) = \dfrac{1}{1+16x^2}$ CG grid $N = 8$: $\|f - I_8^{CG} f\|_\infty \simeq 0.10$



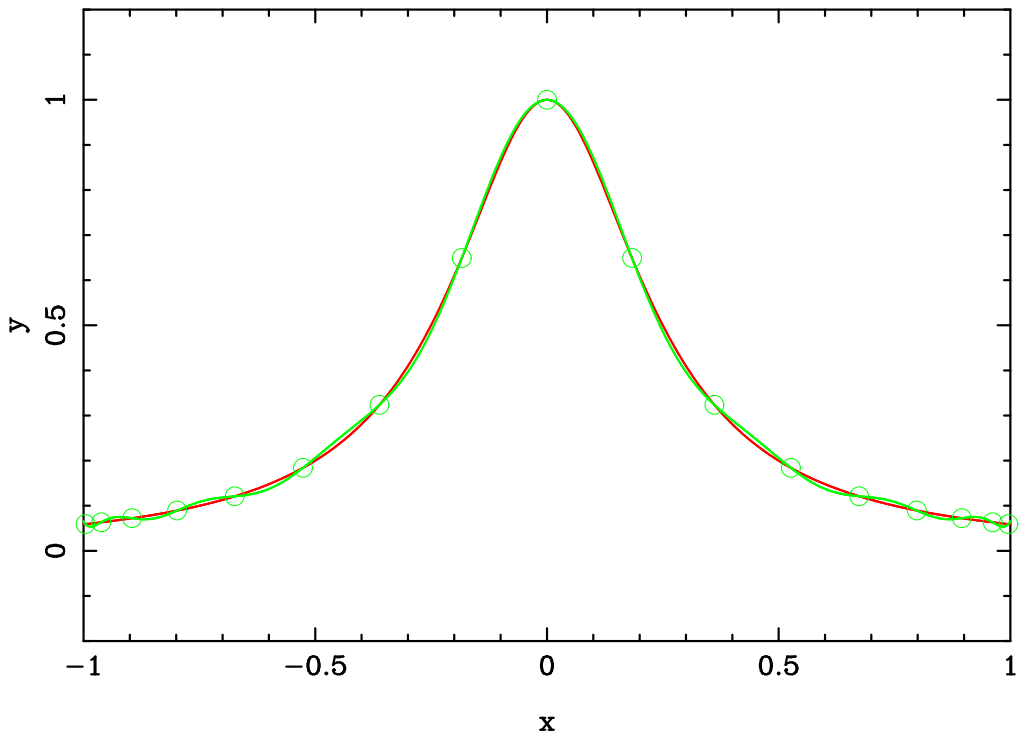
Example: Chebyshev-Gauss interpolation of $f(x) = \frac{1}{1+16x^2}$

$f(x) = \dfrac{1}{1+16x^2}$ CG grid $N = 12$: $\|f - I_{12}^{CG} f\|_\infty \simeq 3.8\ 10^{-2}$



Example: Chebyshev-Gauss interpolation of $f(x) = \frac{1}{1+16x^2}$

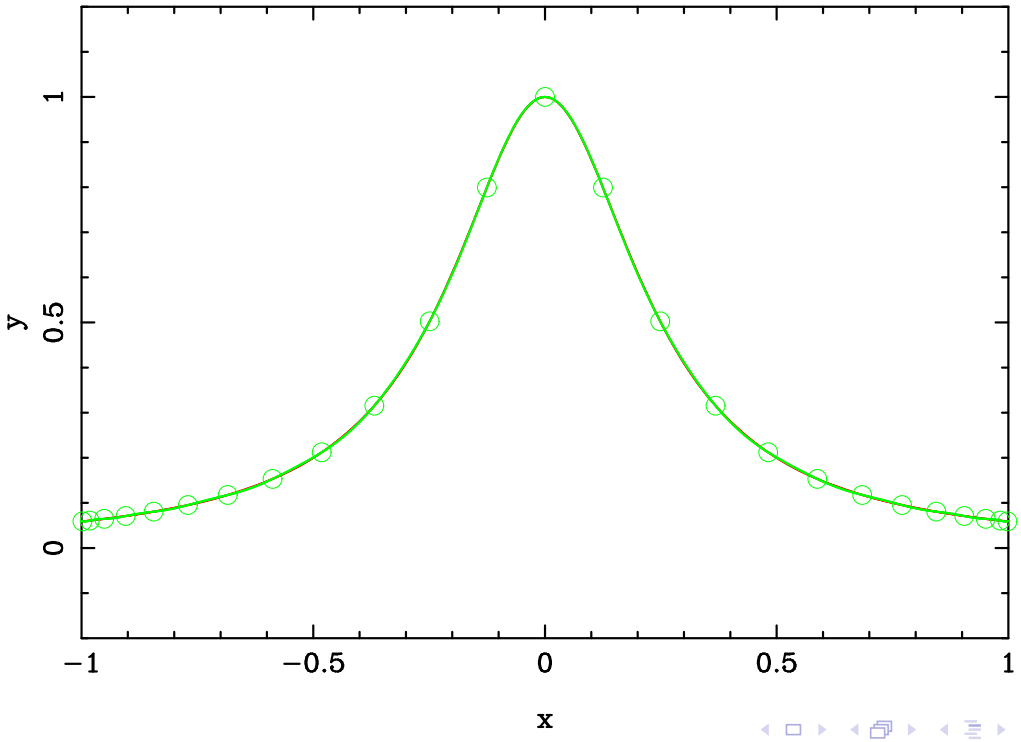
$f(x) = \dfrac{1}{1+16x^2}$ CG grid $N = 16$: $\|f - I_{16}^{CG} f\|_\infty \simeq 1.5\ 10^{-2}$



Example: Chebyshev-Gauss interpolation of $f(x) = \frac{1}{1+16x^2}$

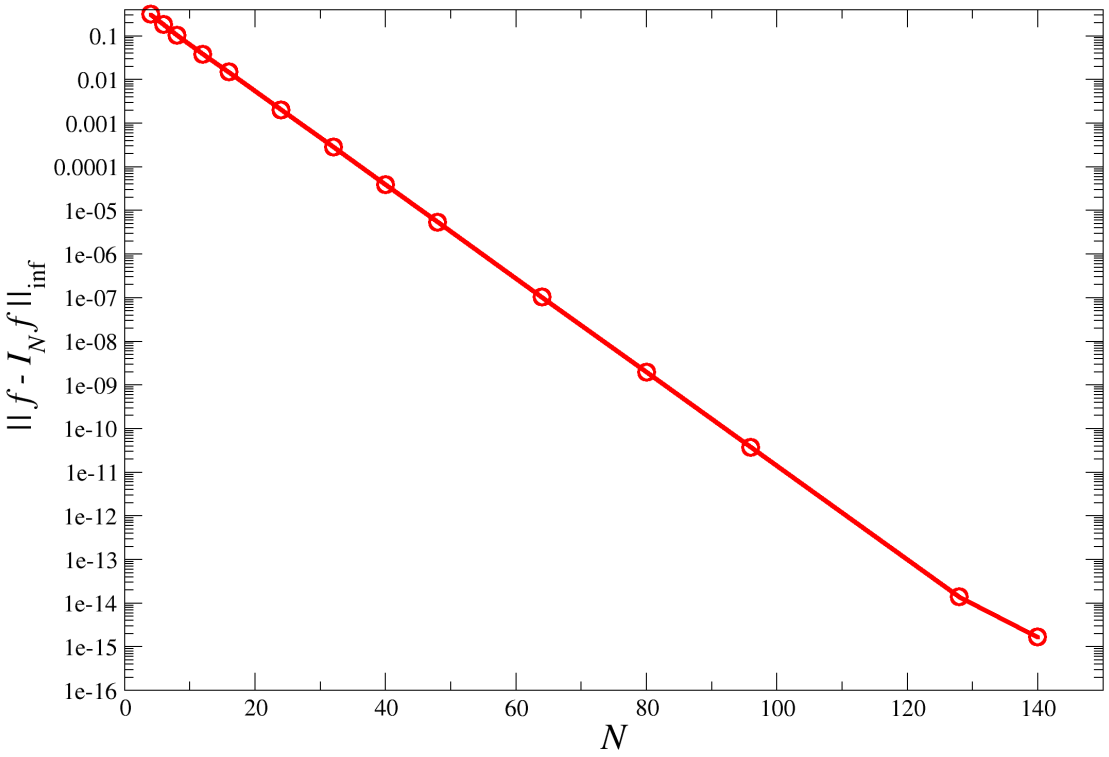
$f(x) = \dfrac{1}{1+16x^2}$ CG grid $N = 24$: $\|f - I_{24}^{CG} f\|_\infty \simeq 2.0\ 10^{-3}$

no Runge phenomenon !



Example: Chebyshev-Gauss interpolation of $f(x) = \frac{1}{1+16x^2}$

Variation of the interpolation error as N increases



Chebyshev polynomials = orthogonal polynomials

The Chebyshev polynomials, the zeros of which provide the Chebyshev-Gauss nodes, constitute a family of **orthogonal polynomials**, and the Chebyshev-Gauss nodes are associated to **Gauss quadratures**.

Outline

Hilbert space $L^2_w(-1,1)$

Framework: Let us consider the functional space

$$L^2_w(-1,1) = \left\{ f : (-1,1) \to \mathbb{R},\ \int_{-1}^{1} f(x)^2\, w(x)\, dx < \infty \right\}$$

where $w : (-1,1) \to (0,\infty)$ is an integrable function, called the weight function.

$L^2_w(-1,1)$ is a **Hilbert space** for the scalar product

$$(f|g)_w := \int_{-1}^{1} f(x)\, g(x)\, w(x)\, dx$$

with the associated norm

$$\|f\|_w := (f|f)_w^{1/2}$$

Orthogonal polynomials

The set $\mathbb{P}$ of polynomials on $[-1,1]$ is a subspace of $L^2_w(-1,1)$.
A family of orthogonal polynomials is a set $(p_i)_{i\in\mathbb{N}}$ such that

- $p_i \in \mathbb{P}$
- $\deg p_i = i$
- $i \neq j \Rightarrow (p_i|p_j)_w = 0$

$(p_i)_{i\in\mathbb{N}}$ is then a basis of the vector space $\mathbb{P}$: $\mathbb{P} = \text{span}\,\{p_i, i \in \mathbb{N}\}$

Theorem

A family of orthogonal polynomial $(p_i)_{i\in\mathbb{N}}$ is a **Hilbert basis** *of $L^2_w(-1,1)$:*

$$\forall f \in L^2_w(-1,1), \quad f = \sum_{i=0}^{\infty} \tilde{f}_i\, p_i \qquad \text{with } \tilde{f}_i := \frac{(f|p_i)_w}{\|p_i\|_w^2}.$$

The above infinite sum means $\lim_{N\to\infty} \left\| f - \sum_{i=0}^{N} \tilde{f}_i\, p_i \right\|_w = 0$

Jacobi polynomials

Jacobi polynomials are orthogonal polynomials with respect to the weight

$$w(x) = (1-x)^\alpha (1+x)^\beta$$

Subcases:

- Legendre polynomials $P_n(x)$: $\alpha = \beta = 0$, i.e. $w(x) = 1$
- Chebyshev polynomials $T_n(x)$: $\alpha = \beta = -\frac{1}{2}$, i.e. $w(x) = \frac{1}{\sqrt{1-x^2}}$

Jacobi polynomials are eigenfunctions of the singular[1] **Sturm-Liouville problem**

$$-\frac{d}{dx}\left[(1-x^2)\,w(x)\,\frac{du}{dx}\right] = \lambda\, w(x)\, u, \qquad x \in (-1,1)$$

[1] *singular* means that the coefficient in front of du/dx vanishes at the extremities of the interval $[-1,1]$

Legendre polynomials

$$w(x) = 1: \quad \int_{-1}^{1} P_i(x) P_j(x)\, dx = \frac{2}{2i+1}\delta_{ij}$$

$P_0(x) = 1$

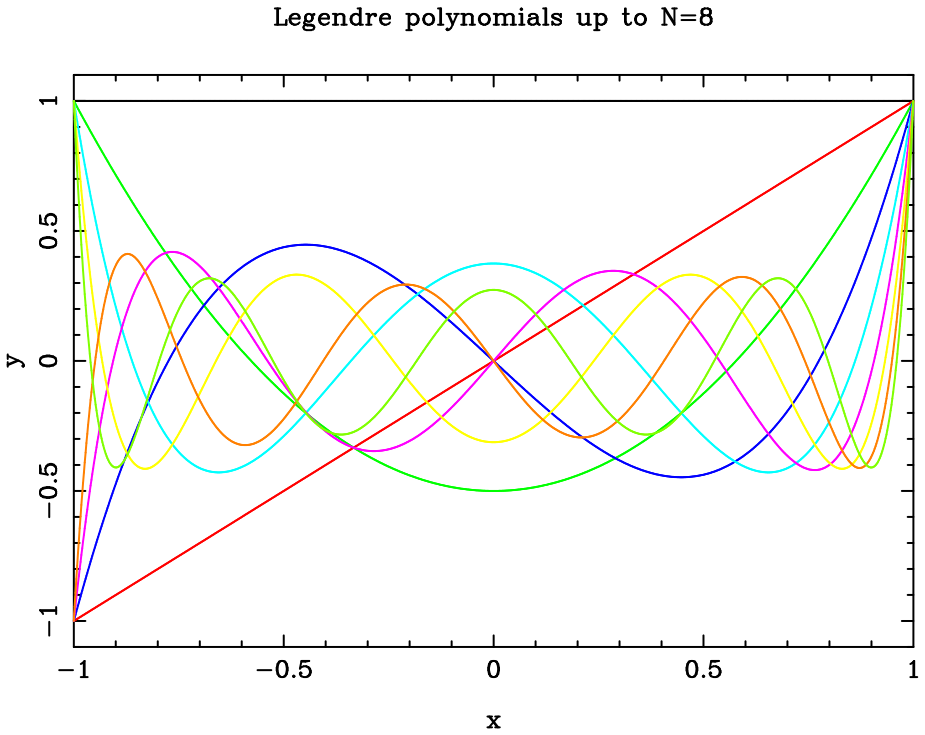
$P_1(x) = x$

$P_2(x) = \frac{1}{2}(3x^2 - 1)$

$P_3(x) = \frac{1}{2}(5x^3 - 3x)$

$P_4(x) = \frac{1}{8}(35x^4 - 30x^2 + 3)$

$P_{i+1}(x) = \frac{2i+1}{i+1} x P_i(x) - \frac{i}{i+1} P_{i-1}(x)$



Chebyshev polynomials

$$w(x) = \frac{1}{\sqrt{1-x^2}}: \quad \int_{-1}^{1} T_i(x) T_j(x) \frac{dx}{\sqrt{1-x^2}} = \frac{\pi}{2}(1+\delta_{0i})\,\delta_{ij}$$

$T_0(x) = 1$

$T_1(x) = x$

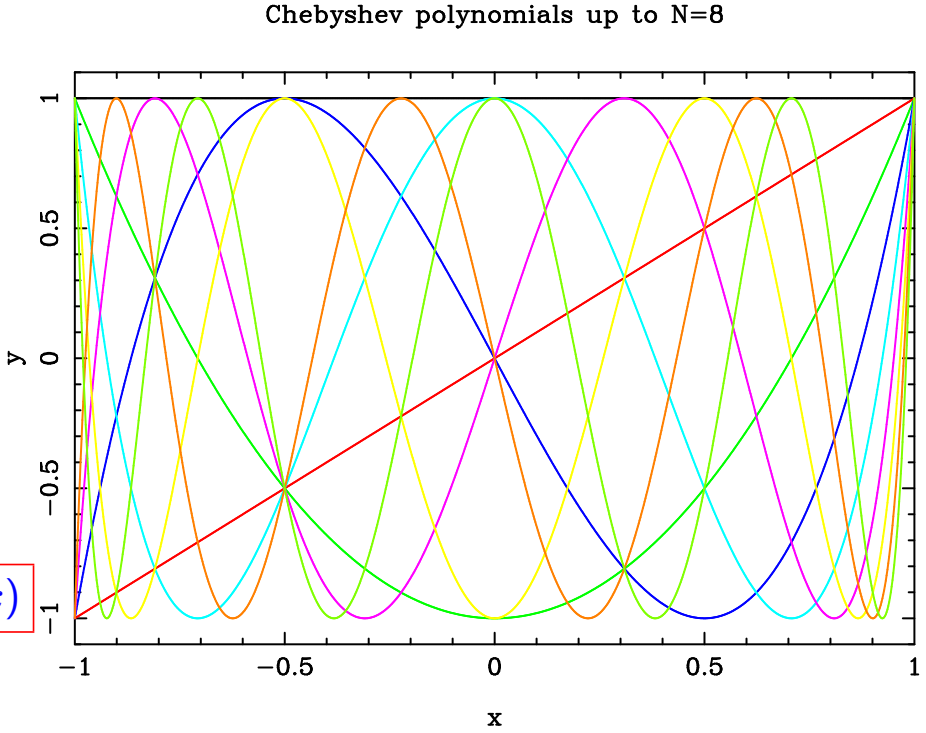
$T_2(x) = 2x^2 - 1$

$T_3(x) = 4x^3 - 3x$

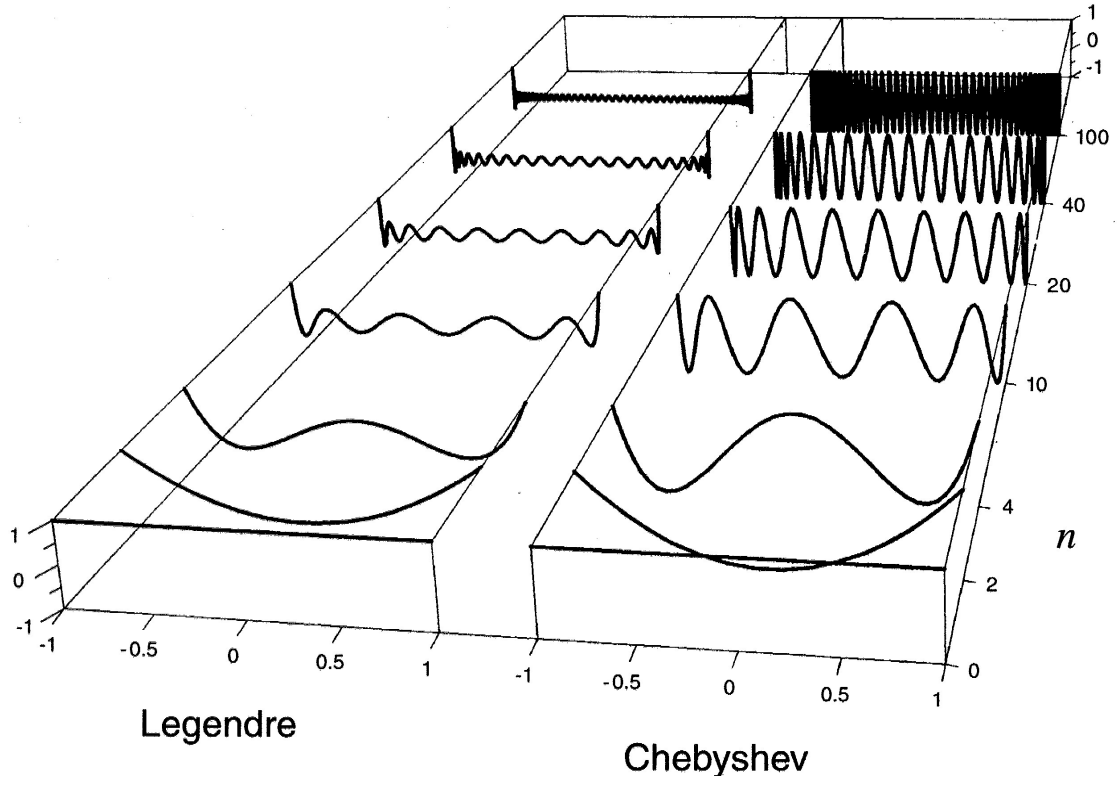
$T_4(x) = 8x^4 - 8x^2 + 1$

$\cos(n\theta) = T_n(\cos\theta)$

$T_{i+1}(x) = 2xT_i(x) - T_{i-1}(x)$



Legendre and Chebyshev compared



[from Fornberg (1998)]

Orthogonal projection on $\mathbb{P}_N$

Let us consider $f \in L^2_w(-1,1)$ and a family $(p_i)_{i\in\mathbb{N}}$ of orthogonal polynomials with respect to the weight w.
Since $(p_i)_{i\in\mathbb{N}}$ is a Hilbert basis of $L^2_w(-1,1)$ reminder
we have $f(x) = \sum_{i=0}^{\infty} \tilde{f}_i\, p_i(x)$ with $\tilde{f}_i := \dfrac{(f|p_i)_w}{\|p_i\|_w^2}$.
The truncated sum

$$\boxed{\Pi_N^w f(x) := \sum_{i=0}^{N} \tilde{f}_i\, p_i(x)}$$

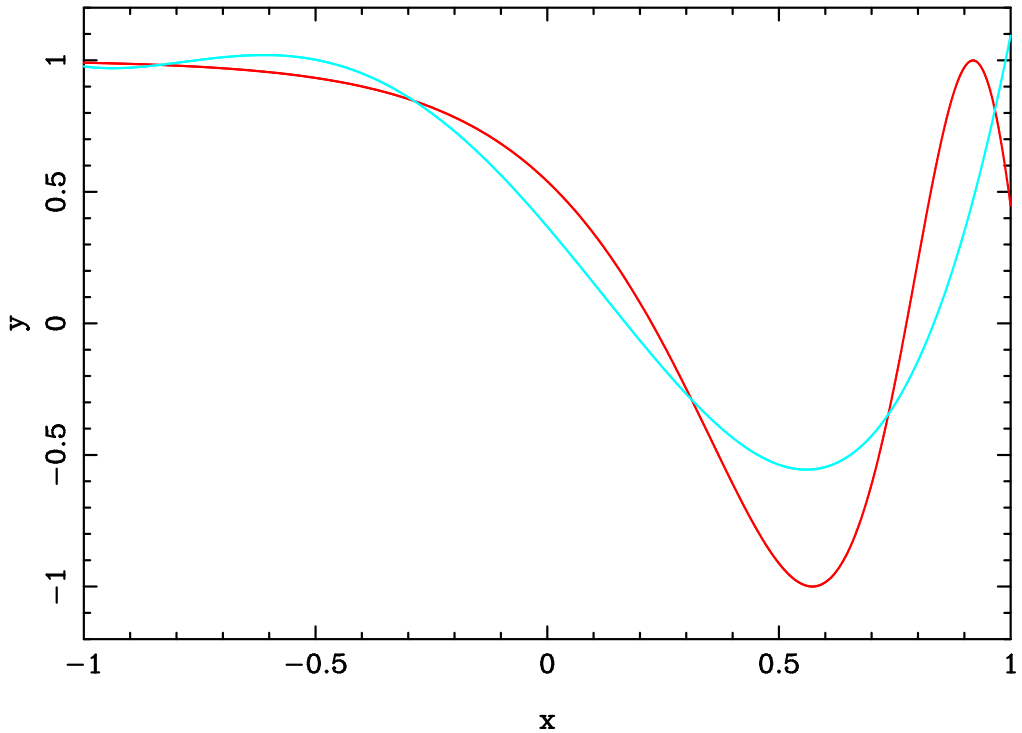
is a polynomial of degree N: it is the orthogonal projection of f onto the finite dimensional subspace $\mathbb{P}_N$ with respect to the scalar product $(.|.)_w$.
We have

$$\boxed{\lim_{N\to\infty} \|f - \Pi_N^w f\|_w = 0}$$

Hence $\Pi_N^w f$ can be considered as a polynomial approximation of the function f.

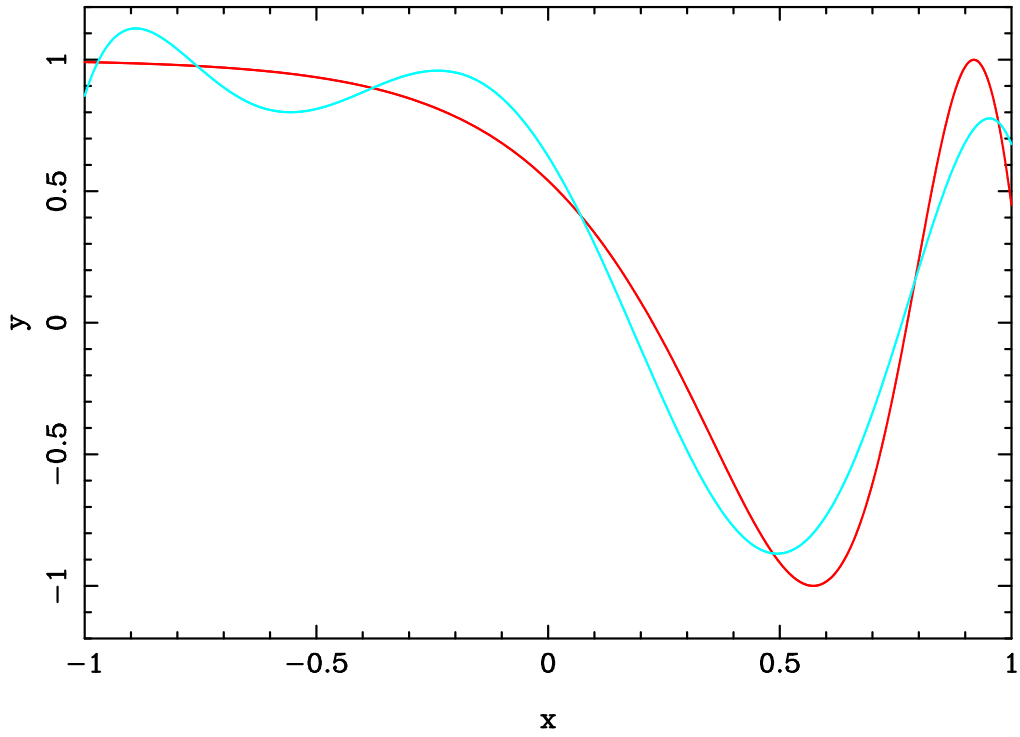
Example: Chebyshev projection of $f(x) = \cos(2\exp(x))$

$f(x) = \cos(2\exp(x))$ $\quad w(x) = (1-x^2)^{-1/2}$ $\quad N = 4$: $\|f - \Pi_4^w f\|_\infty \simeq 0.66$



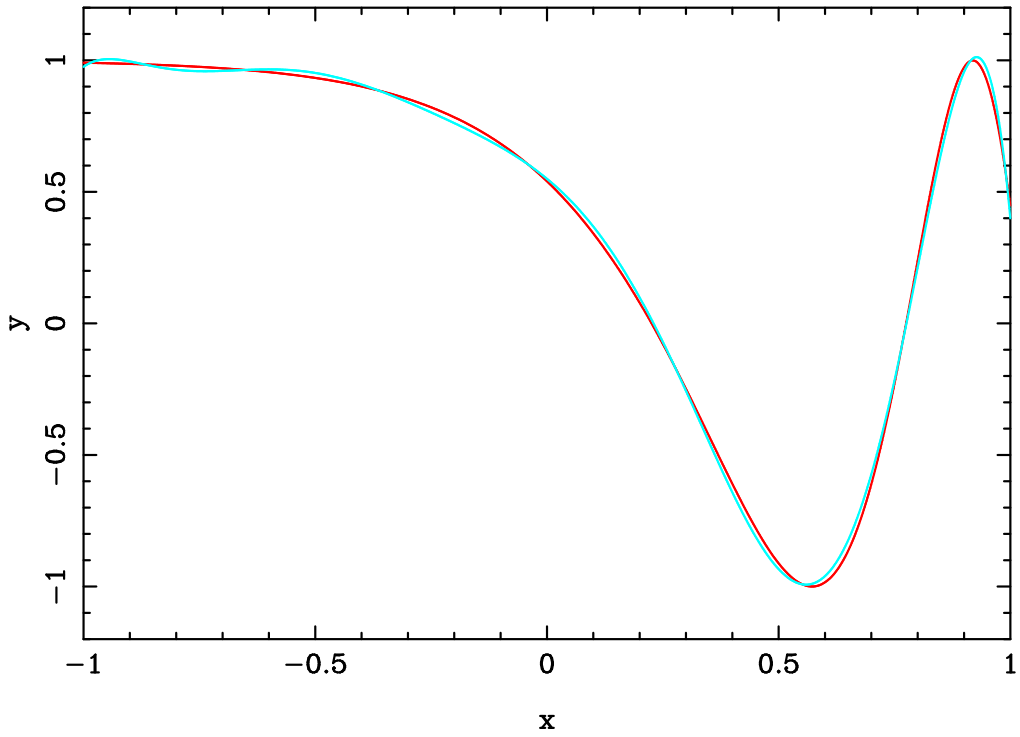
Example: Chebyshev projection of $f(x) = \cos(2\exp(x))$

$f(x) = \cos(2\exp(x))$ $\quad w(x) = (1-x^2)^{-1/2}$ $\quad N = 6 : \|f - \Pi_6^w f\|_\infty \simeq 0.30$



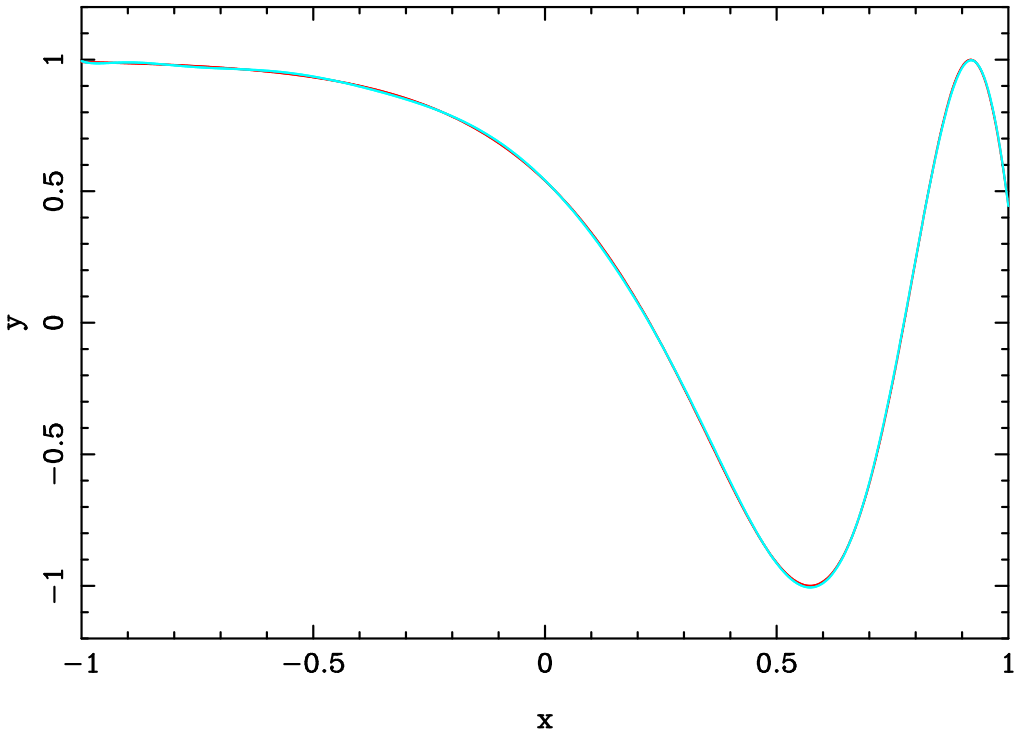
Example: Chebyshev projection of $f(x) = \cos(2\exp(x))$

$f(x) = \cos(2\exp(x))$ $\quad w(x) = (1-x^2)^{-1/2}$ $\quad N = 8 : \|f - \Pi_8^w f\|_\infty \simeq 4.9\ 10^{-2}$



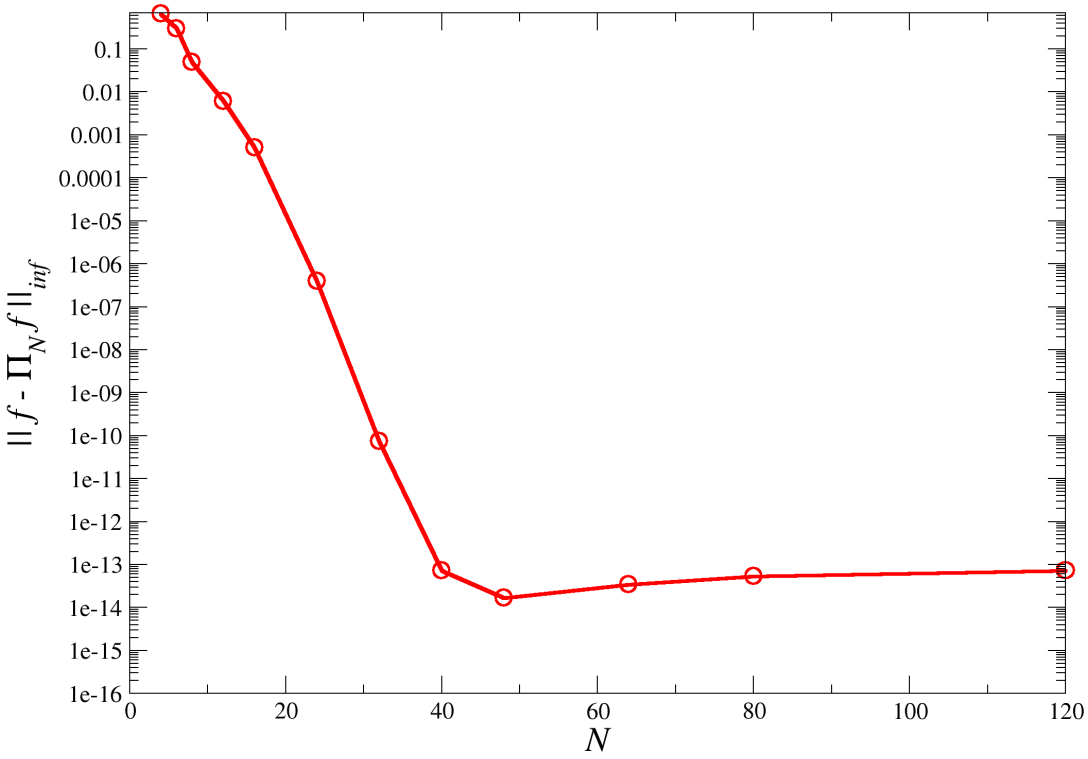
Example: Chebyshev projection of $f(x) = \cos(2\exp(x))$

$f(x) = \cos(2\exp(x))$ $w(x) = (1-x^2)^{-1/2}$ $N = 12$: $\|f - \Pi_{12}^w f\|_\infty \simeq 6.1\ 10^{-3}$



Example: Chebyshev projection of $f(x) = \cos(2\exp(x))$

Variation of the projection error $\|f - \Pi_N^w f\|_\infty$ as N increases



Evaluation of the coefficients

The coefficients $\tilde{f}_i$ of the orthogonal projection of f are given by

$$\tilde{f}_i := \frac{(f|p_i)_w}{\|p_i\|_w^2} = \frac{1}{\|p_i\|_w^2} \int_{-1}^{1} f(x)\, p_i(x)\, w(x)\, dx \qquad (2)$$

Problem: the above integral cannot be computed exactly; we must seek a numerical approximation.

Solution: **Gaussian quadrature**

Gaussian quadrature

Theorem (Gauss, Jacobi)

Let $(p_i)_{i\in\mathbb{N}}$ be a family of orthogonal polynomials with respect to some weight w. For $N > 0$, let $X = (x_i)_{0\leq i\leq N}$ be the grid formed by the $N+1$ zeros of the polynomial p_{N+1} and

$$w_i := \int_{-1}^{1} \ell_i^X(x)\, w(x)\, dx$$

where ℓ_i^X is the i-th Lagrange cardinal polynomial of the grid X reminder
Then

$$\forall f \in \mathbb{P}_{2N+1},\ \int_{-1}^{1} f(x)\, w(x)\, dx = \sum_{i=0}^{N} w_i f(x_i)$$

If $f \notin \mathbb{P}_{2N+1}$, the above formula provides a good approximation of the integral.

Gauss-Lobatto quadrature

The nodes of the Gauss quadrature, being the zeros of p_{N+1}, do not encompass the boundaries -1 and 1 of the interval $[-1, 1]$. For numerical purpose, it is desirable to include these points in the boundaries.

This possible at the price of reducing by 2 units the degree of exactness of the Gauss quadrature

Gauss-Lobatto quadrature

Theorem (Gauss-Lobatto quadrature)

Let $(p_i)_{i\in\mathbb{N}}$ be a family of orthogonal polynomials with respect to some weight w. For $N > 0$, let $X = (x_i)_{0\leq i\leq N}$ be the grid formed by the $N+1$ zeros of the polynomial

$$q_{N+1} = p_{N+1} + \alpha p_N + \beta p_{N-1}$$

where the coefficients α and β are such that $x_0 = -1$ and $x_N = 1$.
Let

$$w_i := \int_{-1}^{1} \ell_i^X(x)\, w(x)\, dx$$

where ℓ_i^X is the i-th Lagrange cardinal polynomial of the grid X.
Then

$$\forall f \in \mathbb{P}_{2N-1},\quad \int_{-1}^{1} f(x)\, w(x)\, dx = \sum_{i=0}^{N} w_i f(x_i)$$

Notice: $f \in \mathbb{P}_{2N-1}$ instead of $f \in \mathbb{P}_{2N+1}$ for Gauss quadrature.

Gauss-Lobatto quadrature

Remark: if the (p_i) are Jacobi polynomials, i.e. if $w(x) = (1-x)^\alpha (1+x)^\beta$, then the Gauss-Lobatto nodes which are strictly inside $(-1,1)$, i.e. $x_1, \ldots, x_{N-1}$, are the $N-1$ zeros of the polynomial p'_N, or equivalently the points where the polynomial p_N is extremal.

This of course holds for Legendre and Chebyshev polynomials.
For Chebyshev polynomials, the Gauss-Lobatto nodes and weights have simple expressions:

$$x_i = -\cos\frac{\pi i}{N}, \quad 0 \leq i \leq N$$

$$w_0 = w_N = \frac{\pi}{2N}, \quad w_i = \frac{\pi}{N}, \quad 1 \leq i \leq N-1$$

Note: in the following, we consider only Gauss-Lobatto quadratures

Discrete scalar product

The Gauss-Lobatto quadrature motivates the introduction of the following scalar product:

$$\boxed{\langle f|g\rangle_N = \sum_{i=0}^{N} w_i f(x_i) g(x_i)}$$

It is called the discrete scalar product associated with the Gauss-Lobatto nodes $X = (x_i)_{0\leq i\leq N}$
Setting $\gamma_i := \langle p_i|p_i\rangle_N$, the discrete coefficients associated with a function f are given by

$$\boxed{\hat{f}_i := \frac{1}{\gamma_i}\langle f|p_i\rangle_N, \qquad 0 \leq i \leq N}$$

which can be seen as approximate values of the coefficients $\tilde{f}_i$ provided by the Gauss-Lobatto quadrature [cf. Eq. (2)]

Discrete coefficients and interpolating polynomial

Let $I_N^{\mathrm{GL}} f$ be the interpolant of f at the Gauss-Lobatto nodes $X = (x_i)_{0 \leq i \leq N}$. Being a polynomial of degree N, it is expandable as

$$I_N^{\mathrm{GL}} f(x) = \sum_{i=0}^{N} a_i \, p_i(x)$$

Then, since $I_N^{\mathrm{GL}} f(x_j) = f(x_j)$,

$$\hat{f}_i = \frac{1}{\gamma_i} \langle f | p_i \rangle_N = \frac{1}{\gamma_i} \langle I_N^{\mathrm{GL}} f | p_i \rangle_N = \frac{1}{\gamma_i} \sum_{j=0}^{N} a_j \langle p_j | p_i \rangle_N$$

Now, if $j = i$, $\langle p_j | p_i \rangle_N = \gamma_i$ by definition. If $j \neq i$, $p_j p_i \in \mathbb{P}_{2N-1}$ so that the Gauss-Lobatto formula holds and gives $\langle p_j | p_i \rangle_N = (p_j | p_i)_w = 0$. Thus we conclude that $\langle p_j | p_i \rangle_N = \gamma_i \delta_{ij}$ so that the above equation yields $\hat{f}_i = a_i$, i.e. **the discrete coefficients are nothing but the coefficients of the expansion of the interpolant at the Gauss-Lobato nodes**

Spectral representation of a function

In a spectral method, the numerical representation of a function f is through its interpolant at the Gauss-Lobatto nodes:

$$\boxed{I_N^{\mathrm{GL}} f(x) = \sum_{i=0}^{N} \hat{f}_i \, p_i(x)}$$

The discrete coefficients $\hat{f}_i$ are computed as

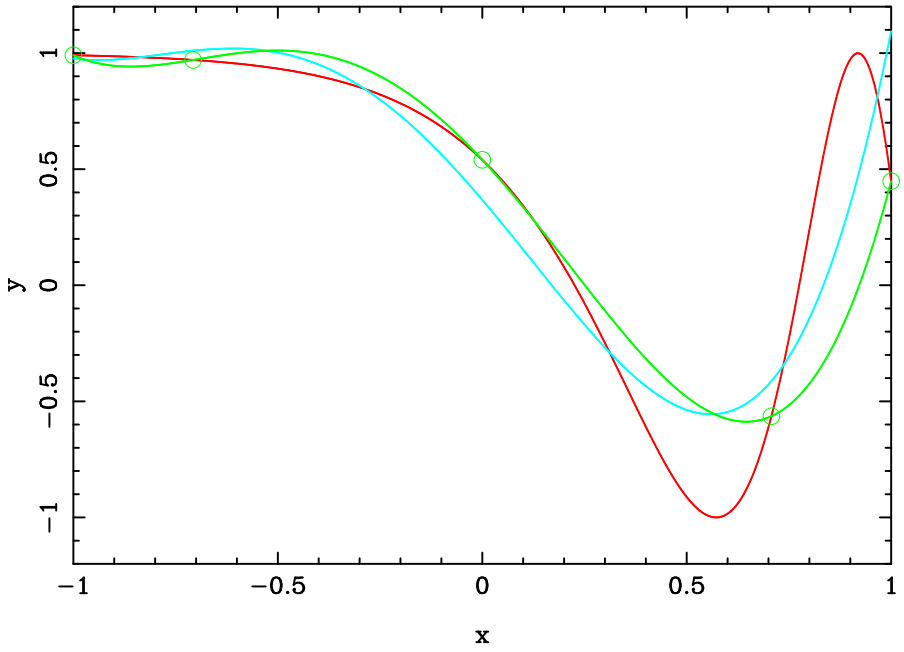
$$\boxed{\hat{f}_i = \frac{1}{\gamma_i} \sum_{j=0}^{N} w_j f(x_j) p_i(x_j)}$$

$I_N^{\mathrm{GL}} f(x)$ is an approximation of the truncated series $\Pi_N^w f(x) = \sum_{i=0}^{N} \tilde{f}_i \, p_i(x)$,

which is the orthogonal projection of f onto the polynomial space $\mathbb{P}_N$.
$\Pi_N^w f$ should be the true spectral representation of f, but in general it is not computable exactly.
The difference between $I_N^{\mathrm{GL}} f$ and $\Pi_N^w f$ is called the aliasing error

Example: aliasing error for $f(x) = \cos(2\exp(x))$

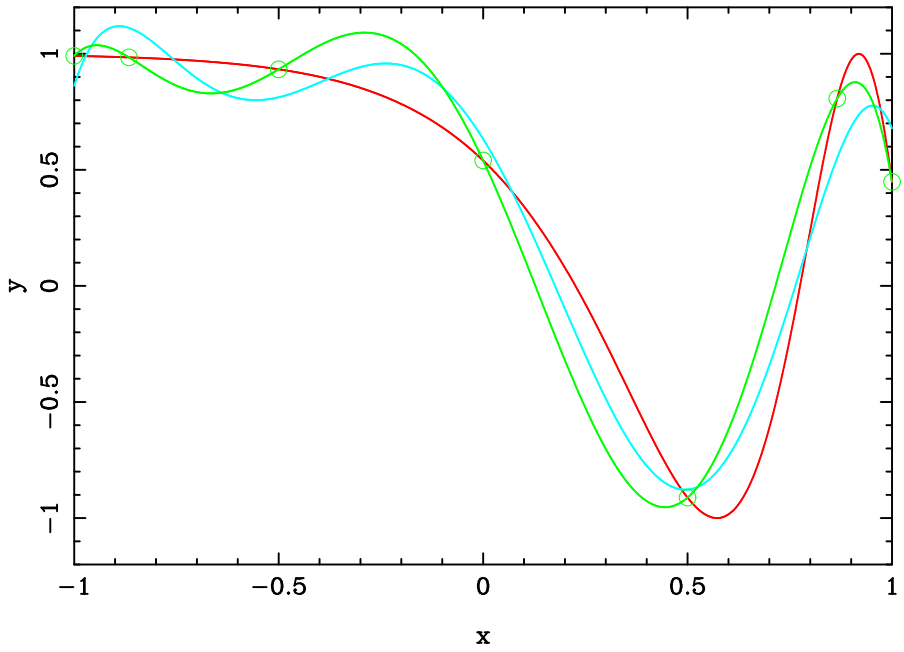
$f(x) = \cos(2\exp(x))$ $\quad w(x) = (1-x^2)^{-1/2}$ $\quad N = 4$



red: f; blue: $\Pi_N^w f$; green: $I_N^{\mathrm{GL}} f$

Example: aliasing error for $f(x) = \cos(2\exp(x))$

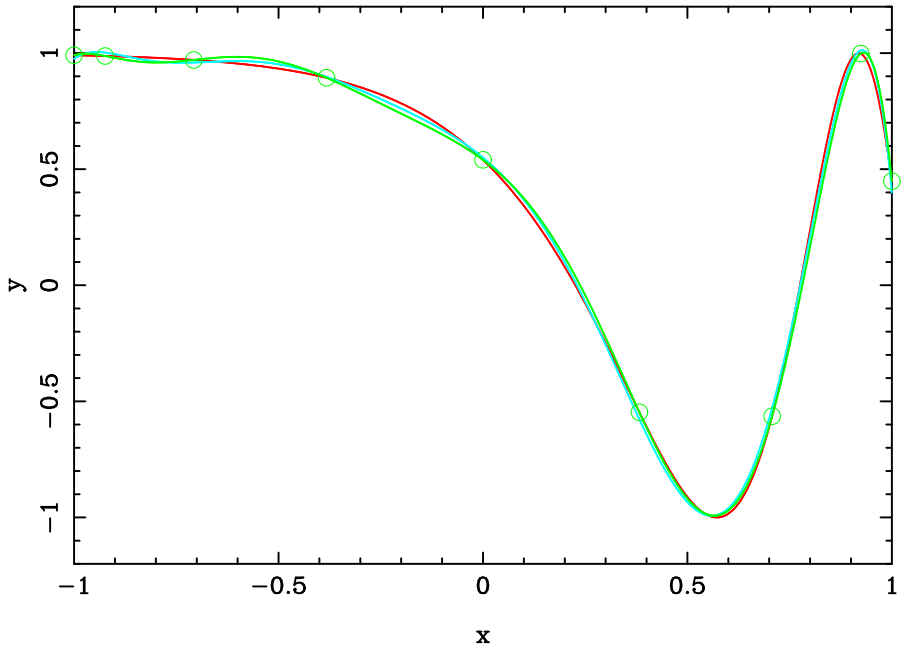
$f(x) = \cos(2\exp(x))$ $\quad w(x) = (1-x^2)^{-1/2}$ $\quad N = 6$



red: f; blue: $\Pi_N^w f$; green: $I_N^{\mathrm{GL}} f$

Example: aliasing error for $f(x) = \cos(2\exp(x))$

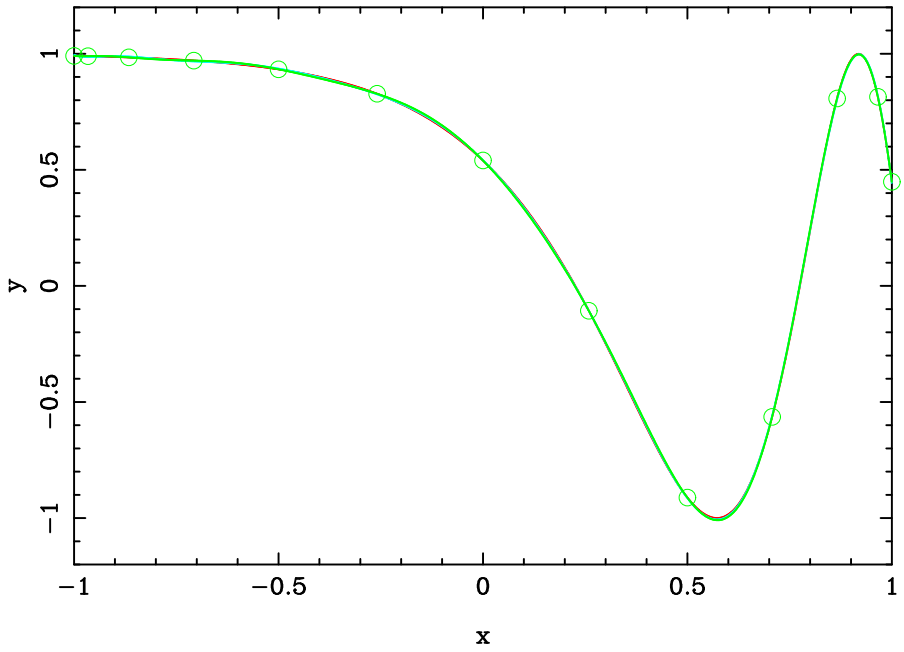
$f(x) = \cos(2\exp(x))$ $\quad w(x) = (1-x^2)^{-1/2}$ $\quad N = 8$



red: f; blue: $\Pi_N^w f$; green: $I_N^{\mathrm{GL}} f$

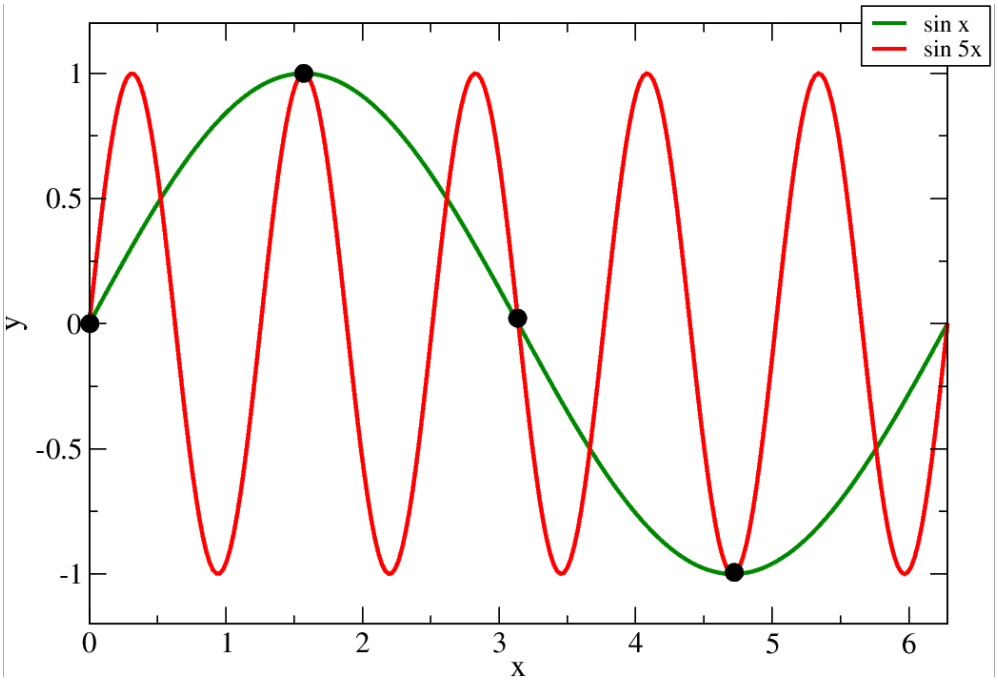
Example: aliasing error for $f(x) = \cos(2\exp(x))$

$f(x) = \cos(2\exp(x))$ $\quad w(x) = (1-x^2)^{-1/2}$ $\quad N = 12$



red: f; blue: $\Pi_N^w f$; green: $I_N^{\mathrm{GL}} f$

Aliasing error = contamination by high frequencies



Aliasing of a $\sin(x)$ wave by a $\sin(5x)$ wave on a 4-points grid

Outline

Sobolev norm

Let us consider a function $f \in C^m([-1,1])$, with $m \geq 0$.

The Sobolev norm of f with respect to some weight function w is

$$\|f\|_{H^m_w} := \left(\sum_{k=0}^{m} \left\| f^{(k)} \right\|_w^2 \right)^{1/2}$$

Convergence rates for $f \in C^m([-1,1])$

Chebyshev expansions:

- truncation error :
  $\|f - \Pi^w_N f\|_w \leq \dfrac{C_1}{N^m} \|f\|_{H^m_w}$ and $\|f - \Pi^w_N f\|_\infty \leq \dfrac{C_2(1+\ln N)}{N^m} \displaystyle\sum_{k=0}^{m} \left\| f^{(k)} \right\|_\infty$
- interpolation error :
  $\left\|f - I^{\mathrm{GL}}_N f\right\|_w \leq \dfrac{C_3}{N^m} \|f\|_{H^m_w}$ and $\left\|f - I^{\mathrm{GL}}_N f\right\|_\infty \leq \dfrac{C_4}{N^{m-1/2}} \|f\|_{H^m_w}$

Legendre expansions:

- truncation error :
  $\|f - \Pi^w_N f\|_w \leq \dfrac{C_1}{N^m} \|f\|_{H^m_w}$ and $\|f - \Pi^w_N f\|_\infty \leq \dfrac{C_2}{N^{m-1/2}} V(f^{(m)})$
- interpolation error :
  $\left\|f - I^{\mathrm{GL}}_N f\right\|_w \leq \dfrac{C_3}{N^{m-1/2}} \|f\|_{H^m_w}$

Evanescent error for smooth functions

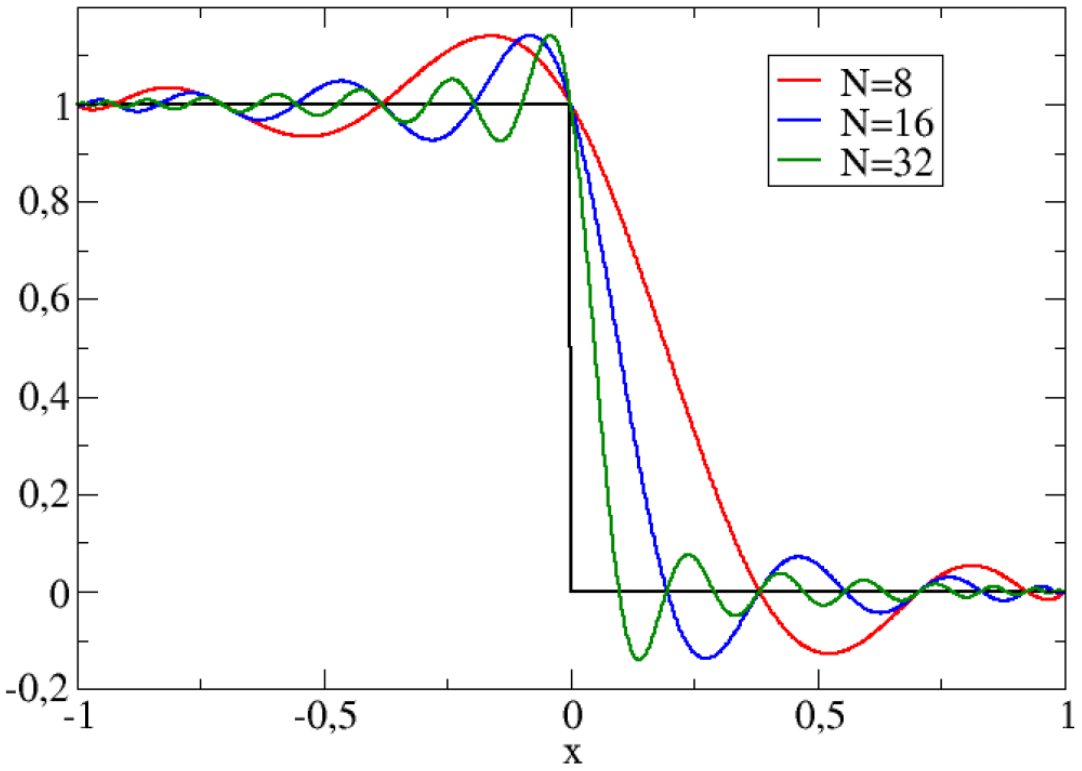
If $f \in C^\infty([-1,1])$, the error of the spectral expansions $\Pi_N^w f$ or $I_N^{\mathrm{GL}} f$ decays more rapidly than any power of N.

In practice: **exponential decay** example

This error is called evanescent.

For non-smooth functions: Gibbs phenomenon

Extreme case: f discontinuous



Outline

References

- C. Bernardi, Y. Maday & F. Rapetti : *Discrétisations variationnelles de problèmes aux limites elliptiques*, Springer (Paris, 2004)
- J.P. Boyd : *Chebyshev and Fourier spectral methods*, Dover (New York, 2001)
- C. Canuto, M.Y. Hussaini, A. Quarteroni & T.A. Zang : *Spectral methods in fluid dynamics*, Springer-Verlag (Berlin, 1988)
- B. Fornberg : *A practical guide to pseudospectral methods*, Cambridge Univ. Press (Cambridge, 1998)
- A. Quarteroni, R. Sacco & F. Saleri : *Méthodes numériques pour le calcul scientifique*, Springer (Paris, 2000)
- M.A. Snyder : *Chebyshev methods in numerical approximation*, Prentice-Hall (Englewood Cliffs, N.J., USA, 1966)
- http://en.wikipedia.org/wiki/Polynomial_interpolation
- http://en.wikipedia.org/wiki/Orthogonal_polynomials

One dimensional PDE

Philippe Grandclément

Laboratoire de l'Univers et de ses Théories (LUTH)
CNRS / Observatoire de Paris
F-92195 Meudon, France

philippe.grandclement@obspm.fr

Collaborators
Silvano Bonazzola, Eric Gourgoulhon, Jérôme Novak

November 14-18, 2005

Outline

INTRODUCTION

Type of problems

We will consider a differential equation :

$$Lu(x) = S(x) \qquad x \in U \tag{1}$$
$$Bu(y) = 0 \qquad y \in \partial U \tag{2}$$

where L are B are linear differential operators.
In the following, we will only consider one-dimensional cases $U = [-1; 1]$.
We will also assume that u can be expanded on some functions :

$$\tilde{u}(x) = \sum_{n=0}^{N} \tilde{u}_n \phi_n(x). \tag{3}$$

Depending on the choice of expansion functions ϕ_k, one can generate :

- finite difference methods.
- finite element method.
- spectral methods.

The weighted residual method

Given a scalar product on U, one makes the residual $R = Lu - S$ small in the sense :

$$\forall k \in \{0, 1, ...N\}, \quad (\xi_k, R) = 0, \tag{4}$$

under the constraint that u verifies the boundary conditions.
The ξ_k are called the test functions.

Standard spectral methods

The expansion functions are global orthogonal polynomials functions, like Chebyshev and Legendre.
Depending on the choice of test functions :

Tau method

The ξ_k are the expansion functions. The boundary conditions are enforced by an additional set of equations.

Collocation method

The $\xi_k = \delta(x - x_k)$ and the boundary conditions are enforced by an additional set of equations.

Galerkin method

The expansions and the test functions are chosen to fulfill the boundary conditions.

Optimal methods

Definition :

A numerical method is said to be optimal iff the resolution of the equation does not introduce an error greater than the one already done by interpoling the exact solution.

- u_{exact} is the exact solution.
- $I_N u_{\text{exact}}$ is the interpolant of the exact solution.
- $u_{\text{num.}}$ is the numerical solution.

The method is optimal iff $\max_\Lambda \left(|u_{\text{exact}} - I_N u_{\text{exact}}|\right)$ and $\max_\Lambda \left(|u_{\text{exact}} - u_{\text{num.}}|\right)$ have the same behavior when $N \longrightarrow \infty$.

ONE-DOMAIN METHODS

Matrix representation of L

The action of L on u can be given by a matrix L_{ij}

If $u = \sum_{k=0}^{N} \tilde{u}_k T_k$ then

$$Lu = \sum_{i=0}^{N} \sum_{j=0}^{N} L_{ij} \tilde{u}_j T_i$$

L_{ij} is obtained by knowing the basis operation on the expansion basis. The k^{th} column is the coefficients of LT_k.

Example of elementary operations with Chebyshev

If $f = \sum_{n=0}^{\infty} a_n T_n(x)$ then $Hf = \sum_{n=0}^{\infty} b_n T_n(x)$

H is the multiplication by x

$$b_n = \frac{1}{2}\left(\left(1 + \delta_{0n-1}\right) a_{n-1} + a_{n+1}\right) \text{ with } n \geq 1$$

H is the derivation

$$b_n = \frac{2}{\left(1 + \delta_{0n}\right)} \sum_{p=n+1, p+n \text{ odd}}^{\infty} p a_p$$

H is the second derivation

$$b_n = \frac{1}{\left(1 + \delta_{0n}\right)} \sum_{p=n+2, p+n \text{ even}}^{\infty} p\left(p^2 - n^2\right) a_p$$

Tau method

The test functions are the T_k

$(T_k|R) = 0$ implies : $\sum_{j=0}^{N} L_{kj}\tilde{u}_j = \tilde{s}_k$ ($N+1$ equations).

The $\tilde{s}_k$ are the coefficients of the interpolant of the source.

Boundary conditions

- $u(x=-1) = 0 \Longrightarrow \sum_{j=0}^{N} (-1)^j \tilde{u}_j = 0$
- $u(x=+1) = 0 \Longrightarrow \sum_{j=0}^{N} \tilde{u}_j = 0$

One considers the $N-1$ first residual equations and the 2 boundary conditions. The unknowns are the $\tilde{u}_k$.

Collocation method

The test functions are the $\delta_k = \delta(x - x_k)$

$(\delta_n|R) = 0$ implies that : $Lu(x_n) = s(x_n)$ ($N+1$ equations).

$$\sum_{i=0}^{N}\sum_{j=0}^{N} \tilde{u}_j L_{ij} T_i(x_n) = s(x_n) \quad \forall n \in [0, N]$$

Boundary conditions

- Like for the Tau-method they are enforced by two additional equations.
- One has to relax the residual conditions in x_0 and x_N.

Galerkin method : choice of basis

We need a set of functions that

- are easily given in terms of basis functions.
- fulfill the boundary conditions.

Example

If one wants $u(-1) = 0$ and $u(1) = 0$, one can choose :

- $G_{2k}(x) = T_{2k+2}(x) - T_0(x)$
- $G_{2k+1}(x) = T_{2k+3}(x) - T_1(x)$

Let us note that only $N-1$ functions G_i must be considered to maintain the same order of approximation (general feature).

Transformation matrix

Definition

The G_i are given in terms of the T_i by a transformation matrix M
M is a matrix of size $N+1 \times N-1$.

$$G_i = \sum_{j=0}^{N} M_{ji} T_j \quad \forall i \leq N-2 \tag{5}$$

Example

$$M_{ij} = \begin{pmatrix} -1 & 0 & -1 \\ 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The Galerkin system (1)

Expressing the equations$(G_n|R)$

- u is expanded on the Galerkin basis.

$$u = \sum_{i=0}^{N-2} \tilde{u}_i^G G_i(x). \tag{6}$$

- The expression of Lu is obtained in terms of T_i via M_{ij} and L_{ij}.
- $(G_n|Lu)$ is computed by using, once again M_{ij}
- The source is NOT expanded in terms of G_i but by the T_i.
- $(G_n|S)$ is obtained by using M_{ij}
- This is $N-1$ equations.

The Galerkin system (2)

$(G_n|R) = 0 \quad \forall n \leq N-2$

$$\sum_{k=0}^{N-2} \tilde{u}_k^G \sum_{i=0}^{N} \sum_{j=0}^{N} M_{in} M_{jk} L_{ij} (T_i|T_i) = \sum_{i=0}^{N} M_{in} \tilde{s}_i (T_i|T_i), \quad \forall n \leq N-2 \tag{7}$$

The $N-1$ unknowns are the coefficients $\tilde{u}_n^G$.
The transformation matrix M is then used to get :

$$u(x) = \sum_{k=0}^{N} \left(\sum_{n=0}^{N-2} M_{kn} \tilde{u}_n^G \right) T_k$$

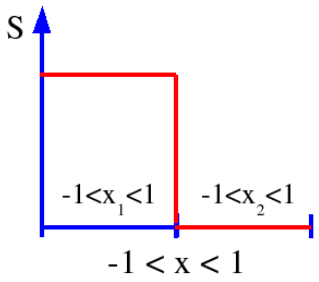
MULTI-DOMAIN METHODS

Multi-domain decomposition

Motivations

- We have seen that discontinuous functions (or not $\mathcal{C}^\infty$ functions) are not well represented by spectral expansion.
- However, in physics, we may be interested in such fields (for example the surface of a strange star can produce discontinuities).
- We also may need to use different functions in various regions of space.
- In order to cope with that, we need several domains in such a way that the discontinuities lies at the boundaries.
- By doing so, the functions are $\mathcal{C}^\infty$ in every domain, preserving the exponential convergence.

Multi-domain setting



- $x = \frac{1}{2}(x_1 - 1)$
- $x = \frac{1}{2}(x_2 + 1)$

Spectral decomposition with respect to x_i

- Domain 1 : $u(x < 0) = \sum_{i=0}^{N} \tilde{u}_i^1 T_i(x_1(x))$
- Domain 2 : $u(x > 0) = \sum_{i=0}^{N} \tilde{u}_i^2 T_i(x_2(x))$
- Same thing for the source.

Note that $\frac{\mathrm{d}}{dx} = 2\frac{\mathrm{d}}{dx_i}$

A multi-domain Tau method

Domain 1

- $(T_k|R) = 0 \Longrightarrow \sum_{j=0}^{N} L_{kj}\tilde{u}_j^1 = \tilde{s}_k^1$
- $N+1$ equations and we relax the last two. (N-1 equations)
- Same thing in domain 2.

Additional equations :

- the 2 boundary conditions.
- matching of the solution at $x = 0$.
- matching of the first derivative at $x = 0$.

A complete system

- 2N-2 equations for residuals and 4 for the matching and boundary conditions.
- 2N+2 unknowns, the $\tilde{u}_i^1$ and $\tilde{u}_i^2$

Homogeneous solution method

This method is the closest to the standard analytical way of solving linear differential equations.

Principle

- find a particular solution in each domain.
- compute the homogeneous solutions in each domain.
- determine the coefficients of the homogeneous solutions by imposing :
  - the boundary conditions.
  - the matching of the solution at the boundary.
  - the matching of the first derivative.

Homogeneous solutions

In general 2 in each domain and they can be known either :

- by numerically solving $Lu = 0$.
- or, most of the time, they can be found analytically.

The number of homogeneous solutions can be modified for regularity reasons.

Particular solution

In each domain, we can seek a particular solution g by a Tau residual method.

$$(T_k|R) = 0 \Longrightarrow \sum_{j=0}^{N} L_{kj}\tilde{g}_j = \tilde{s}_k$$

However, due to the presence of homogeneous solutions, the matrix L_{ij} is degenerate.
More precisely, L_{ij} is more and more degenerate as $N \to \infty$, the homogeneous solution being better described by their interpolant.

$$\sum_{j=0}^{N} L_{kj}\tilde{h}_j \to 0 \text{ when } N \to \infty$$

The non-degenerate operator

A non-degenerate operator O can be obtained by removing :

- the m first columns of L_{ij} (imposes that the first m coefficients of g are 0).
- the m last lines of L_{ij} (relaxes the last m equations for the residual).
- m is the number of homogeneous solutions (typically $m = 2$).

The matrix O is, generally, non-degenerate, and can be inverted.(true as long as the m first coefficients of the HS are not 0...)

Matching system

Example

- 2 domains.
- 2 homogeneous solutions in each of them.

The system (4 equations)

- two boundary conditions (left and right).
- matching of the solution across the boundary.
- matching of the first radial derivative.

The unknowns are the coefficients of the homogeneous solutions (4 in this particular case).

Variational formulation

Warning : this method is easily applicable only when using Legendre polynomials because it requires that $w(x) = 1$.
We will write Lu as $Lu \equiv -u'' + Fu$, F being a first order differential operator on u.

Starting point

- weighted residual equation :

$$(\xi|R) = 0 \Longrightarrow \int \xi \left(-u'' + Fu\right) \mathrm{d}x = \int \xi s \mathrm{d}x$$

- Integration by part :

$$[-\xi u'] + \int \xi' u' \mathrm{d}x + \int \xi F u \mathrm{d}x = \int \xi s \mathrm{d}x$$

Test functions

As for the collocation method : $\xi = \delta_k = \delta(x - x_k)$ for all points but $x = -1$ and $x = 1$.

Various operators

Derivation in configuration space

$$g'(x_k) = \sum_{j=0}^{N} D_{kj} g(x_j) \tag{8}$$

First order operator F in the configuration space

$$Fu(x_k) = \sum_{j=0}^{N} F_{kj} u(x_j) \tag{9}$$

Expression of the integrals

$[-\xi u'] + \int \xi' u' \mathrm{d}x + \int \xi F u \mathrm{d}x = \int \xi s \mathrm{d}x$

- $\int \xi_n s \mathrm{d}x = \sum_{i=0}^{N} \xi_n(x_i)\, s(x_i)\, w_i = s(x_n)\, w_n$
- $\int \xi_n F u \mathrm{d}x = \sum_{i=0}^{N} \xi_n(x_i)\, Fu(x_i)\, w_i = \left[\sum_{j=0}^{N} F_{nj} u(x_j) \right] w_n$
- $\int \xi'_n u' \mathrm{d}x = \sum_{i=0}^{N} \xi'_n(x_i)\, u'(x_i)\, w_i = \sum_{i=0}^{N} \sum_{j=0}^{N} D_{ij} D_{in} w_i u(x_j)$

Equations for the points inside the domains

$[-\xi u'] = 0$ so that, in each domain :

$$\sum_{i=0}^{N}\sum_{j=0}^{N} D_{ij}D_{in}w_i u(x_j) + \left[\sum_{j=0}^{N} F_{nj}u(x_j)\right] w_n = s(x_n)\, w_n$$

In each domain : $0 < n < N$, i.e. 2N-2 equations.

Equations at the boundary

In the domain 1 :

$n = N$ and $[-\xi u'] = -u'^1(x_1 = 1; x = 0)$

$$u'^1(x_1 = 1) = \sum_{i=0}^{N}\sum_{j=0}^{N} D_{ij}D_{iN}w_i u^1(x_j) + \left[\sum_{j=0}^{N} F_{Nj}u^1(x_j)\right] w_N - s^1(x_N)\, w_N$$

In the domain 2 :

$n = 0$ and $[-\xi u'] = u'^2(x_2 = -1; x = 0)$

$$u'^2(x_2 = -1) = -\sum_{i=0}^{N}\sum_{j=0}^{N} D_{ij}D_{i0}w_i u^2(x_j) - \left[\sum_{j=0}^{N} F_{0j}u^2(x_j)\right] w_0 + s^2(x_0)\, w_0$$

Matching equation

$u'^1(x_1 = 1; x = 0) = u'^2(x_2 = -1; x = 0) \Longrightarrow$

$$\sum_{i=0}^{N}\sum_{j=0}^{N} D_{ij}D_{iN}w_i u^1(x_j) + \left[\sum_{j=0}^{N} F_{Nj}u^1(x_j)\right] w_N + \sum_{i=0}^{N}\sum_{j=0}^{N} D_{ij}D_{i0}w_i u^2(x_j) + \left[\sum_{j=0}^{N} F_{0j}u^2(x_j)\right] w_0 = s^1(x_N)\, w_N + s^2(x_0)\, w_0$$

Additional equations

- Boundary condition at $x = -1$: $u^1(x_0) = 0$
- Boundary condition at $x = 1$: $u^2(x_N) = 0$
- Matching at $x = 0$: $u^1(x_N) = u^2(x_0)$

We solve for the unknowns $u^i(x_j)$.

Why Legendre ?

Suppose we use Chebyshev : $w(x) = \dfrac{1}{\sqrt{1-x^2}}$.

$$\int -u'' f w \mathrm{d}x = [-u' f w] + \int u' f' w' \mathrm{d}x$$

Difficult (if not impossible) to compute u' at the boundary, given that w is divergent there $\Longrightarrow$ difficult to impose the weak matching condition.

SOME LORENE OBJECTS

Array of `double` : the `Tbl`

- Constructor : `Tbl::Tbl(int ... )`. The number of dimension is 1, 2 or 3.
- Allocation : `Tbl::set_etat_qcq()`
- Allocation to zero : `Tbl::annule_hard()`
- Reading of an element : `Tbl::operator()(int ...)`
- Writing of an element : `Tbl::set(int...)`
- Output : operator `cout`

Matrix : `Matrice`

- Constructor : `Matrice::Matrice(int, int)`.
- Allocation : `Matrice::set_etat_qcq()`
- Allocation to zero : `Matrice::annule_hard()`
- Reading of an element : `Matrice::operator()(int, int)`
- Writing of an element : `Matrice::set(int, int)`
- Output : operator `cout`
- Allocation of the banded form : `Matrice::set(int up, int down)`
- Computes the LU decomposition : `Matrice::set_lu()`
- Inversion of a system $AX = Y$: `Tbl Matrice::inverse(Tbl y)`. The LU decomposition must be done before.

Tuesday directory

What it provides

- Routines to computes collocation points, weights, and coefficients (using Tbl).
- For Chebyshev (cheby.h and cheby.C)
- For Legendre (leg.h and leg.C)
- The action of the second derivative in Chebyshev space (solver.C)

What should I do ?

- Go to `Lorene/School05` directory.
- type `cvs update -d` to get todays files.
- compile `solver` (using make).
- run it ... (disappointing isnt'it ?).
- write what is missing.

I. A TEST PROBLEM

We propose to solve a simple 1D problem, using a single domain. Let us consider the following equation:

$$u'' - 4u' + 4u = \exp(x) + C \quad ; \quad x \in [-1;1] \quad ; \quad C = -\frac{4e}{1+e^2}. \tag{1}$$

For the boundary conditions, we adopt :

$$u(-1) = 0 \quad \text{and} \quad u(1) = 0. \tag{2}$$

Under those conditions, the solution of the problem is

$$u(x) = \exp(x) - \frac{\sinh 1}{\sinh 2}\exp(2x) + \frac{C}{4}. \tag{3}$$

II. SUGGESTED STEPS

- Construct the matrix representation of the differential operator.
- Solve the equation using one or more of usual methods : Tau, collocation and Galerkin.
- Check whether the methods are optimal or not.

III. DISCONTINUOUS SOURCE

Let us consider the following problem :

$$-u'' + 4u = S \quad ; \quad x \in [-1;1] \tag{4}$$
$$u(-1) = 0 \quad ; \quad u(1) = 0 \tag{5}$$
$$S(x<0) = 1 \quad ; \quad S(x>0) = 0 \tag{6}$$

The solution is given by :

$$u(x<0) = \frac{1}{4} - \left(\frac{e^2}{4} + B^- e^4\right)\exp(2x) + B^-\exp(-2x) \tag{7}$$
$$u(x>0) = B^+\left(\exp(-2x) - \frac{1}{e^4}\exp(2x)\right) \tag{8}$$
$$B^- = -\frac{1}{8(1+e^2)} - \frac{e^2}{8(1+e^4)} \tag{9}$$
$$B^+ = \frac{e^4}{8}\left(\frac{e^2}{(1+e^4)} - \frac{1}{(1+e^2)}\right) \tag{10}$$

IV. SUGGESTED STEPS

- Verify that Gibbs phenomenon appear when using a single domain method.
- Implement one or more if the multi-domain solvers (Tau, Homogeneous solutions or variationnal).
- Check that exponential convergence to the solution is recovered.

Spectral methods in Lorene: regularity, symmetries, operators, ...

Jérôme Novak

Jerome.Novak@obspm.fr

Laboratoire de l'Univers et de ses Théories (LUTH)
CNRS / Observatoire de Paris, France

in collaboration with
Éric Gourgoulhon & Philippe Grandclément

November, 16 2005

Lorene presentation
Jérôme Novak
Introduction
History
General points
Regularity
Spherical coordinates
Analicity
Spectral bases
Symmetries
Spectral representation in Lorene
Mg3d
Multigrid arrays
Base_val and Valeur
Mappings
Scalar field implementation
Important methods
dzpuis flag
Finite part
Diff
Vector fields

Plan

Motivations and history

The numerical library Lorene (for Langage Objet pour la RElativité NumériquE) was initiated in 1997 by **Jean-Alain Marck** who, after realizing that the FORTRAN programming language the group has been using until then, was no longer adapted to the growing complexity of the numerical relativity codes.

Lorene is of course :

- a modular library written in C++,
- a collaborative effort (over 20 contributors),
- many users across the world,
- many published results in numerical relativity ...

thanks to :

- the `cvs` repository,
- fully-documented sources available on the web page `http ://www.lorene.obspm.fr`,
- a great effort to achieve portability across various systems / compilers.

Observatoire de Paris LUTH
Laboratoire de l'Univers et de ses Théories

Common features for many classes

Lorene presentation
Jérôme Novak
Introduction
History
General points
Regularity
Spherical coordinates
Analicity
Spectral bases
Symmetries
Spectral representation in Lorene
Mg3d
Multigrid arrays
Base_val and Valeur
Mappings
Scalar field implementation
Important methods
dzpuis flag
Finite part
Diff
Vector fields

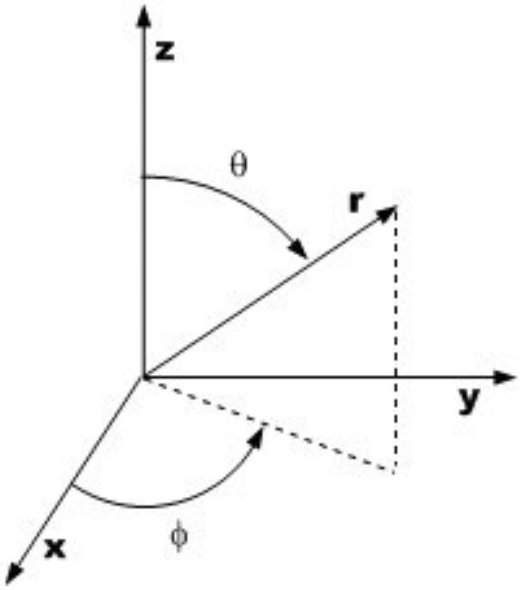
Most of classes (object types) in Lorene share some common functionalities :

- protected data, with readonly accessors often called `.get_XXX` and read/write accessors `.set_XXX`,
- an overload of the "<<" operator to display objects,
- a method for saving data into files and a constructor from a file,
- for container-like objects (arrays, fields...) a state (etat in French) flag indicating whether memory has been allocated :
  - `ETATQCQ` : ordinary state, memory allocated ⇒`set_etat_qcq()` ;
  - `ETATZERO` : null state, memory not allocated ⇒`set_etat_zero()` ;
  - `ETATNONDEF` : undefined state, memory not allocated ⇒`set_etat_nondef()` ;
  - + a method `annule_hard()` to fill with 0s ;
- external arithmetic operators (+, -, *, /) and mathematical functions (sin, exp, sqrt, abs, max, ...).

3D spherical polar coordinates

In almost all cases, fields are represented using 3D spherical coordinates r, θ, φ and a spherical-like grid :

- stars and black holes have spherical shapes,
- astrophysical systems are isolated : boundary conditions are defined for $r \longrightarrow \infty$
- although spherical coordinates are singular (origin, z-axis), surfaces $r =$ constant are smooth.



$$\begin{aligned} x &= r \sin\theta \cos\varphi \\ y &= r \sin\theta \sin\varphi \\ z &= r \cos\theta, \end{aligned}$$

Observatoire de Paris LUTH Laboratoire de l'Univers et de ses Théories

LORENE presentation
Jérôme Novak
Introduction
History
General points
Regularity
Spherical coordinates
Analicity
Spectral bases
Symmetries
Spectral representation in LORENE
Mg3d
Multigrid arrays
Base_val and Valeur
Mappings
Scalar field implementation
Important methods
dzpuis flag
Finite part
Diff
Vector fields

Regularity at the origin

Let $f(x, y, z)$ be an analytic function, it can be expanded near the origin in terms of Taylor series :

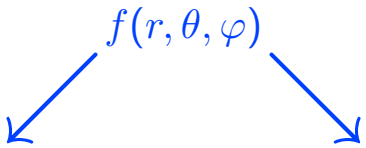
$$f(x, y, z) = \sum_{i=0}^{\infty} \sum_{j=0}^{\infty} \sum_{k=0}^{\infty} c_{ijk} x^i y^j z^k.$$

Changing the coordinates to spherical ones and after some amount of calculations and recasting $\cos\varphi$ and $\sin\varphi$ in $e^{i\varphi}$:

$$f(r, \theta, \varphi) = \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} r^\ell \sum_{i=0}^{\infty} a_{i\ell m} r^{2i} Y_\ell^m(\theta, \varphi).$$

Here $Y_\ell(\theta, \varphi) = P_\ell^m(\cos(\theta))\, e^{im\varphi}$ are the spherical harmonics with $P_\ell^m(\cos(\theta))$ being an associated Legendre polynomial in $\cos\theta$.

Spectral bases



ℓ EVEN		
Radial base	θ base	φ base
Even Chebyshev	Even Fourier	Fourier
Even Chebyshev	Even Legendre	Fourier

ℓ ODD		
Radial base	θ base	φ base
Odd Chebyshev	Odd Fourier	Fourier
Odd Chebyshev	Odd Legendre	Fourier

- Fourier series in θ $\Rightarrow$computation of derivatives or $1/\sin\theta$ operators ;
- associated Legendre polynomial in $\cos\theta$ $\Rightarrow$spherical harmonics $\Rightarrow$computation of the angular Laplace operator

$$\Delta_{\theta\varphi} \equiv \frac{\partial^2}{\partial\theta^2} + \frac{1}{\tan\theta}\frac{\partial}{\partial\theta} + \frac{1}{\sin^2\theta}\frac{\partial^2}{\partial\varphi^2}$$

and inversion of the Laplace or d'Alembert operators.

Observatoire de Paris LUTH Laboratoire de l'Univers et de ses Théories

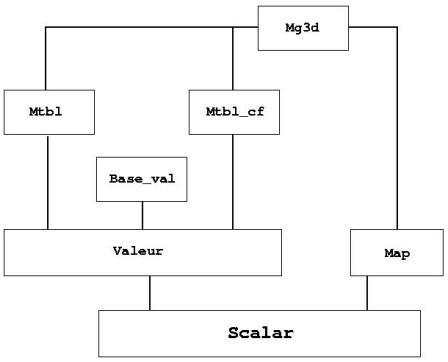
Lorene presentation
Jérôme Novak
Introduction
History
General points
Regularity
Spherical coordinates
Analicity
Spectral bases
Symmetries
Spectral representation in Lorene
Mg3d
Multigrid arrays
Base_val and Valeur
Mappings
Scalar field implementation
Important methods
dzpuis flag
Finite part
Diff
Vector fields

Additional symmetries can be taken into account :

- the θ-*symmetry* : symmetry with respect to the equatorial plane ($z = 0$) ;
- the φ-*symmetry* : invariance under the $(x, y) \mapsto (-x, -y)$ transform.

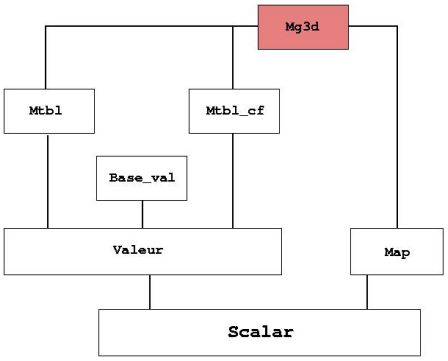
When required, only the angular functions which satisfy these symmetries are used for the decomposition and the grid is reduced in size.
The regularity condition on the z-axis is automatically taken into account by the spherical harmonics basis.

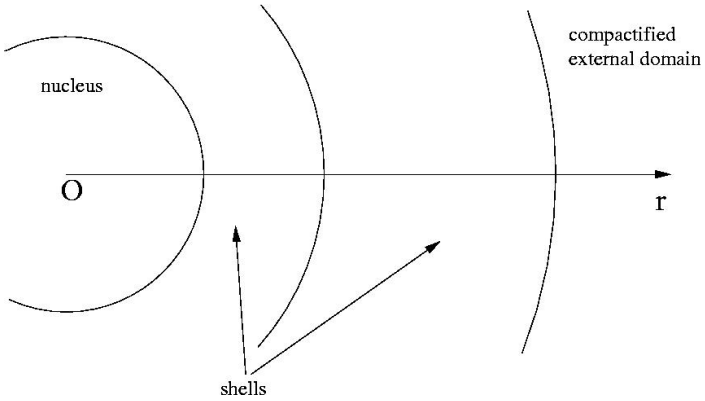
Spectral representation in Lorene



Lorene presentation
Jérôme Novak
Introduction
History
General points
Regularity
Spherical coordinates
Analicity
Spectral bases
Symmetries
Spectral representation in Lorene
Mg3d
Multigrid arrays
Base_val and Valeur
Mappings
Scalar field implementation
Important methods
dzpuis flag
Finite part
Diff
Vector fields

Mg3d

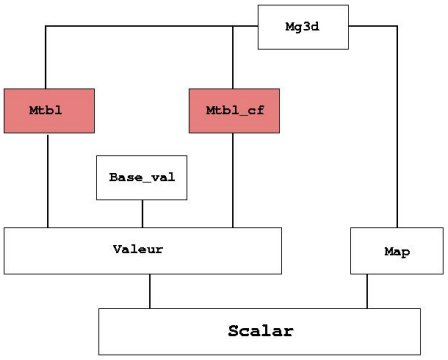




Multi-domain grid of collocation points on which the functions are evaluated to compute the spectral coefficients. It takes into account symmetries.

In each domain, the radial variable used is $\xi \in [-1, 1]$, or $\in [0, 1]$ for the nucleus.

Multigrid arrays

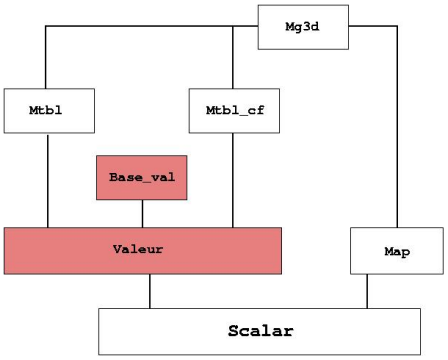


- The class `Mtbl` stores values of a function on grid points; it depends on a multi-domain grid of type `Mg3d` and is merely a collection of 3D arrays `Tbl`.
- The class `Mtbl_cf` stores spectral coefficients of a function; it has two more elements than the corresponding `Mtbl` in the φ-direction.

Observatoire de Paris LUTH
Laboratoire de l'Univers et de ses Théories

Base_val AND Valeur

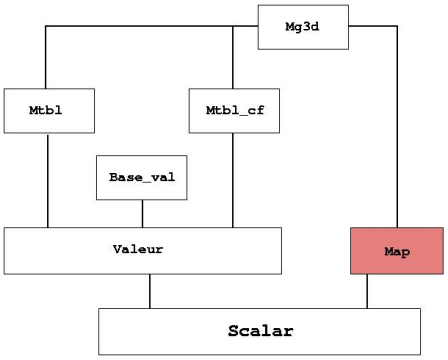
Lorene presentation
Jérôme Novak
Introduction
History
General points
Regularity
Spherical coordinates
Analicity
Spectral bases
Symmetries
Spectral representation in Lorene
Mg3d
Multigrid arrays
Base_val and Valeur
Mappings
Scalar field implementation
Important methods
dzpuis flag
Finite part
Diff
Vector fields

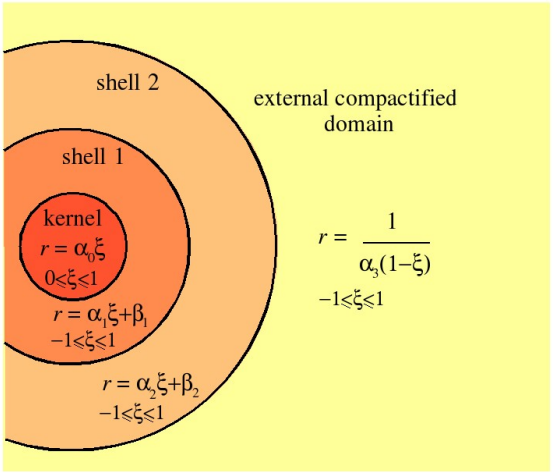


- The class `Base_val` contains information about the spectral bases used in each domain to transform from the function values on the grid points (`Mtbl`) to the spectral coefficients (`Mtbl_cf`).
- The class `Valeur` gathers a `Mtbl`, a `Mtbl_cf` and the `Base_val` to pass from one to the other.

An object of type `Valeur` can be initialized through its `Mtbl` (physical space); the coefficients can then be computed using the method `coef()` or `ylm()` for Fourier or spherical harmonics angular bases. The inverse methods are `coef_i()` and `ylm_i()`.

Mappings
Class Map_af



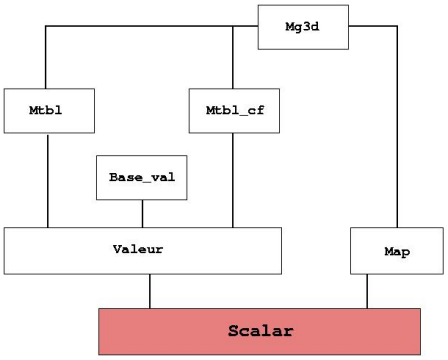


A mapping relates, in each domain, the numerical grid coordinates (ξ, θ', φ') to the physical ones (r, θ, φ).

The simplest class is `Map_af` for which the relation between ξ and r is linear (nucleus + shells) or inverse (CED).

To a mapping are attached coordinate fields `Coord` :
$r, \theta, \varphi, x, y, z, \cos\theta, \cdots$; vector orthogonal triads and flat metrics.

Scalar fields
Class Scalar



The class `Scalar` gathers a `Valeur` and a mapping, it represents a scalar field defined on the spectral grid, or a component of a vector/tensor.

A way to construct a `Scalar` is to

1. use the standard constructor, which needs a mapping ; the associated `Valeur` being then constructed in an undefined state (`ETATNONDEF` ;
2. assign it an expression using `Coords` : *e.g.* `x*y + exp(z)`.

Important methods of the class Scalar

Accessors and modifier of the Valeur

- `get_spectral_va()` readonly
- `set_spectral_va()` read/write ; it can be used to compute spectral coefficients, or to access directly to the coefficients (`Mtbl_cf`).

Spectral base manipulation

- `std_spectral_base()` sets the standard spectral base for a scalar field ;
- `std_spectral_base_odd()` sets the spectral base for the radial derivative of a scalar field ;
- `get_spectral_base()` returns the `Base_val` of the considered `Scalar` ;
- `set_spectral_base(Base_val)` sets a given `Base_val` as the spectral base.

Observatoire de Paris LUTH Laboratoire de l'Univers et de ses Théories

Important methods of the class Scalar

Lorene presentation
Jérôme Novak
Introduction
History
General points
Regularity
Spherical coordinates
Analicity
Spectral bases
Symmetries
Spectral representation in Lorene
Mg3d
Multigrid arrays
Base_val and Valeur
Mappings
Scalar field implementation
Important methods
dzpuis flag
Finite part
Diff
Vector fields

Accessors and modifier of values in a given domain

- `domain(int)` reading ;
- `set_domain(int)` modifying ; it can be used to change the values in the physical space in one domain only.

Accessors and modifier of values of a grid point

- `val_grid_point(int, int, int, int)` readonly in the physical space ;
- `set_grid_point(int, int, int, int)` read/write in the physical space, but should be used with caution, read carefully the documentation.

The dzpuis flag

In the compactified external domain (CED), the variable $u = 1/r$ is used (up to a factor α). $\Rightarrow$when computing the radial derivative (*i.e.* using the method `dsdr()`) of a field f, one gets

$$\frac{\partial f}{\partial u} = -r^2 \frac{\partial f}{\partial r}.$$

For the inversion Laplace operator, since

$$\Delta_r = u^4 \Delta_u,$$

it is interesting to have the source multiplied by r^4 in the CED.
$\Rightarrow$use of an integer flag `dzpuis` for a scalar field f, which means that in the CED, one does not have f, but

$$r^{\text{dzpuis}} f$$

stored.
For instance, if f is constant equal to one in the CED, but with a `dzpuis` set to 4, it means that $\boxed{f = 1/r^4}$ in the CED.

Observatoire de Paris LUTH Laboratoire de l'Univers et de ses Théories

Regular operators and finite part

Lorene presentation
Jérôme Novak
Introduction
History
General points
Regularity
Spherical coordinates
Analicity
Spectral bases
Symmetries
Spectral representation in Lorene
Mg3d
Multigrid arrays
Base_val and Valeur
Mappings
Scalar field implementation
Important methods
dzpuis flag
Finite part
Diff
Vector fields

An operator like $1/r^2$ is singular, in general, at the origin.
Nevertheless, when it appears within *e.g.* the Laplace operator

$$\Delta = \frac{\partial^2}{\partial r^2} + \frac{2}{r}\frac{\partial}{\partial r} + \frac{1}{r^2}\Delta_{\theta\varphi}$$

it should give regular results, when applied to a regular field.
$\Rightarrow$parity $+ r^\ell$ behavior near the origin ensure that everything is well behaved...in theory !
In practice, numerical errors can make things diverge if the division by r is performed in the physical space.
$\Rightarrow$these kind of operators are evaluated in the coefficient space, resulting in

$$\frac{1}{r} \leftrightarrow \frac{f(r) - f(0)}{r}.$$

Operator matrices

All radial operators can be seen, in a given domain, as a matrix multiplication on the vector of Chebyshev coefficients.
The class `Diff` and its derived classes can give directly this matrix :

- there is a different type for each operator
- for example, the second derivative is `Diff_dsdx2`
- standard constructors for all these classes need the number of coefficients and the type of spectral base :

```
Diff_dsdx2 op(17, R_CHEBP) ;
const Matrice mat_op = op.get_matrice() ;
```

Note that this gives the operator with respect to the ξ coordinate...

Observatoire de Paris LUTH Laboratoire de l'Univers et de ses Théories

Vector fields

Lorene can handle a vector field $\boldsymbol{V}$ (class `Vector`) expressed in either of two types of components (*i.e.* using two *orthonormal* triads, of type `Base_vect`) :

- the spherical triad $(V_r, V_\theta, V_\varphi)$ `get_bvect_spher()`,
- the Cartesian triad (V_x, V_y, V_z) `get_bvect_cart()`.

Note that the choice of triad is independent from that of coordinates : one can use $V_y(r, \theta, \varphi)$.

- The Cartesian components of a regular vector field in spherical coordinates follow the same rules that a regular scalar field, except for symmetries ;
- The spherical components have more complicated rules since the spherical triad is singular (additional singularity).

$\Rightarrow$two ways of defining a regular vector field in spherical components :

- define it in Cartesian components and then rotate it (method `change_triad(Base_vect)`), or
- define it as a gradient of a regular scalar field.

Tensor calculus with Lorene

Eric Gourgoulhon

Laboratoire de l'Univers et de ses Théories (LUTH)
CNRS / Observatoire de Paris
F-92195 Meudon, France

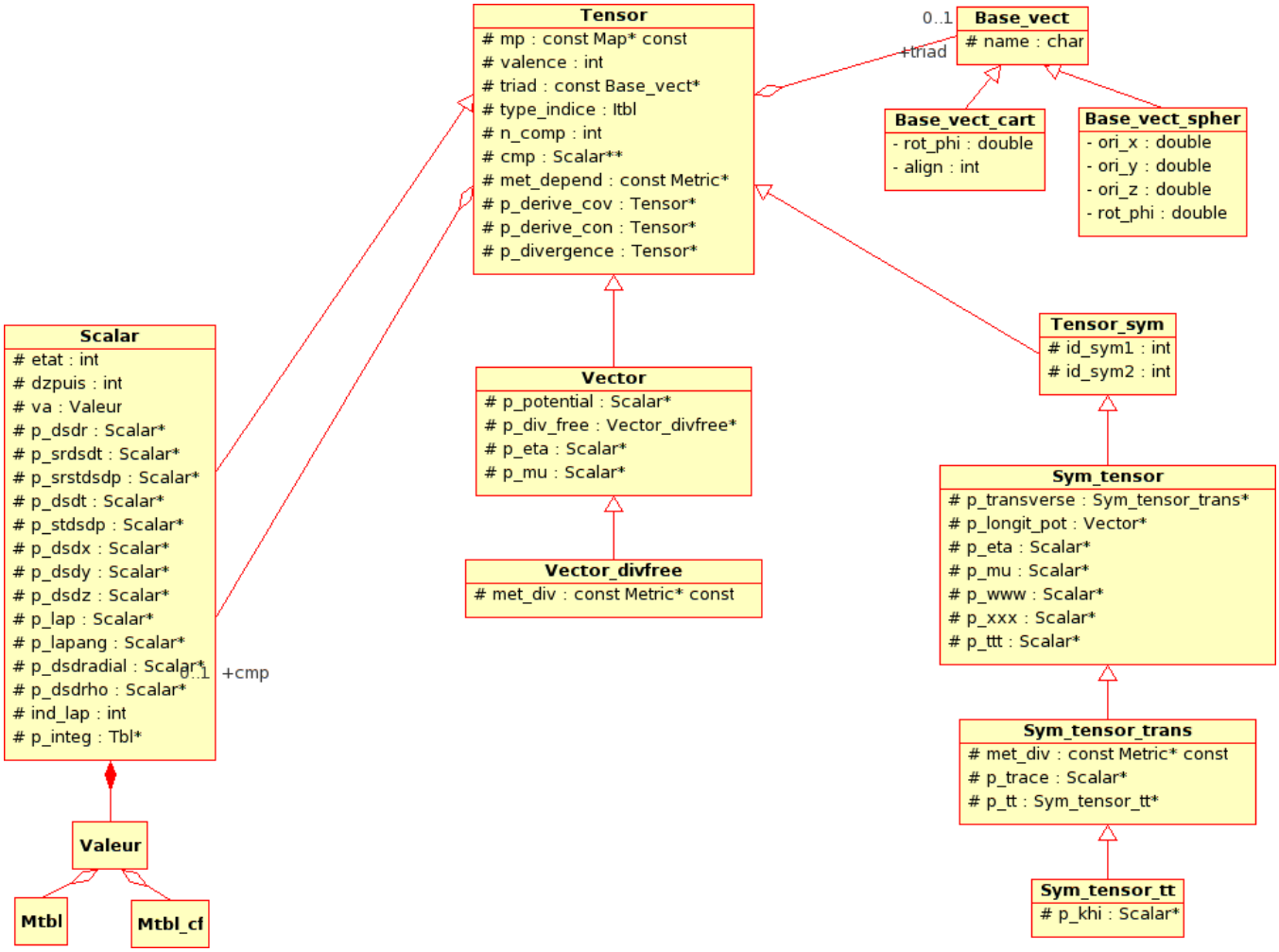
eric.gourgoulhon@obspm.fr

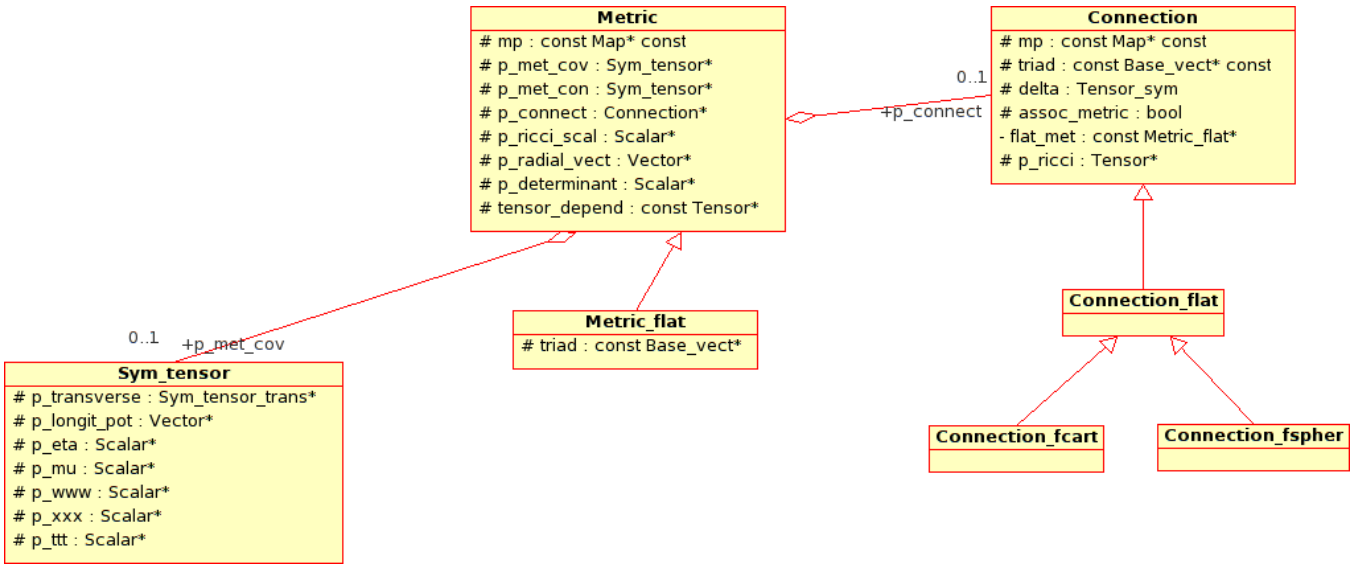
based on a collaboration with
Philippe Grandclément & Jérôme Novak

School on spectral methods:
Application to General Relativity and Field Theory
Meudon, 14-18 November 2005
http://www.lorene.obspm.fr/school/

General features of tensor calculus in LORENE

- Tensor calculus on a 3-dimensional manifold only (3+1 formalism of general relativity)
- Main class: `Tensor` : stores tensor components with respect to a given triad and not *abstract tensors*
- Different metrics can be used at the same time (class `Metric`), with their associated covariant derivatives
- Covariant derivatives can be defined irrespectively of any metric (class `Connection`)
- Dynamical gestion of dependencies guaranties that all quantities are up to date, being recomputed only if necessary


Tensor
mp : const Map* const
valence : int
triad : const Base_vect*
type_indice : Itbl
n_comp : int
cmp : Scalar**
met_depend : const Metric*
p_derive_cov : Tensor*
p_derive_con : Tensor*
p_divergence : Tensor*
0..1
+triad
Base_vect
name : char
Base_vect_cart
- rot_phi : double
- align : int
Base_vect_spher
- ori_x : double
- ori_y : double
- ori_z : double
- rot_phi : double
Scalar
etat : int
dzpuis : int
va : Valeur
p_dsdr : Scalar*
p_srdsdt : Scalar*
p_srstdsdp : Scalar*
p_dsdt : Scalar*
p_stdsdp : Scalar*
p_dsdx : Scalar*
p_dsdy : Scalar*
p_dsdz : Scalar*
p_lap : Scalar*
p_lapang : Scalar*
p_dsdradial : Scalar*
p_dsdrho : Scalar*
ind_lap : int
p_integ : Tbl*
0..1
+cmp
Vector
p_potential : Scalar*
p_div_free : Vector_divfree*
p_eta : Scalar*
p_mu : Scalar*
Vector_divfree
met_div : const Metric* const
Tensor_sym
id_sym1 : int
id_sym2 : int
Sym_tensor
p_transverse : Sym_tensor_trans*
p_longit_pot : Vector*
p_eta : Scalar*
p_mu : Scalar*
p_www : Scalar*
p_xxx : Scalar*
p_ttt : Scalar*
Sym_tensor_trans
met_div : const Metric* const
p_trace : Scalar*
p_tt : Sym_tensor_tt*
Sym_tensor_tt
p_khi : Scalar*
Valeur
Mtbl
Mtbl_cf


Metric
mp : const Map* const
p_met_cov : Sym_tensor*
p_met_con : Sym_tensor*
p_connect : Connection*
p_ricci_scal : Scalar*
p_radial_vect : Vector*
p_determinant : Scalar*
tensor_depend : const Tensor*
0..1
+p_connect
Connection
mp : const Map* const
triad : const Base_vect* const
delta : Tensor_sym
assoc_metric : bool
- flat_met : const Metric_flat*
p_ricci : Tensor*
Metric_flat
triad : const Base_vect*
Connection_flat
0..1
+p_met_cov
Sym_tensor
p_transverse : Sym_tensor_trans*
p_longit_pot : Vector*
p_eta : Scalar*
p_mu : Scalar*
p_www : Scalar*
p_xxx : Scalar*
p_ttt : Scalar*
Connection_fcart
Connection_fspher

Class Base_vect (triads)

The triads are decribed by the LORENE class: `Base_vect`; most of the time, orthonormal triads are used. Two triads are naturally provided, in relation to the coordinates (r, θ, φ) (described by the class `Map`):

- $(\boldsymbol{e}_x, \boldsymbol{e}_y, \boldsymbol{e}_z) = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right)$ (class `Base_vect_cart`)
- $(\boldsymbol{e}_r, \boldsymbol{e}_\theta, \boldsymbol{e}_\varphi) = \left(\frac{\partial}{\partial r}, \frac{1}{r} \frac{\partial}{\partial \theta}, \frac{1}{r \sin\theta} \frac{\partial}{\partial \varphi} \right)$ (class `Base_vect_spher`)

Notice that both triads are orthonormal with respect to the flat metric metric $f_{ij} = \mathrm{diag}(1, 1, 1)$.
Given a coordinate system, described by a mapping (class `Map`), they are obtainable respectively by the methods

- `Map::get_bvect_cart()`
- `Map::get_bvect_spher()`

Class Tensor (tensorial fields)

Conventions: the indices of the tensor components, vary between 1 and 3.
In the example $T^i{}_{jk}$, the first index i is called index no. 0, the second index j is called index no. 1, etc...
The covariance type of the indices is indicated by an integer which takes two values, defined in file `tensor.h`:

- `COV` : covariant index
- `CON` : contravariant index

The covariance types are stored in an array of integers (LORENE class `Itbl`) of size the tensor valence. For $T^i{}_{jk}$, the `Itbl`, `tipe` say, has a size of 3 and is such that

- `tipe(0) = CON`
- `tipe(1) = COV`
- `tipe(2) = COV`

An example of code

This code is available as
Lorene/School05/Wednesday/demo_tensor.C
in the LORENE *distribution*

```
// C headers
#include <stdlib.h>
#include <assert.h>
#include <math.h>

// Lorene headers
#include "headcpp.h"    // standard input/output C++ headers
                        // (iostream, fstream)
#include "metric.h"     // classes Metric, Tensor, etc...
#include "nbr_spx.h"    // defines __infinity as an ordinary number
#include "graphique.h"  // for graphical outputs
#include "utilitaires.h"    // utilities

int main() {
```



```
    // Setup of a multi-domain grid (Lorene class Mg3d)
    // ------------------------------------------------
    int nz = 3 ;  // Number of domains
    int nr = 17 ;  // Number of collocation points in r in each doma
    int nt = 9 ;  // Number of collocation points in theta in each c
    int np = 8 ;  // Number of collocation points in phi in each dom
    int symmetry_theta = SYM ; // symmetry with respect to the
                               // equatorial plane
    int symmetry_phi = NONSYM ; // no symmetry in phi
    bool compact = true ; // external domain is compactified

    // Multi-domain grid construction:
    Mg3d mgrid(nz, nr, nt, np, symmetry_theta, symmetry_phi,
               compact) ;

    cout << mgrid << endl ;
```

```
// Setup of an affine mapping : grid --> physical space
// (Lorene class Map_af)
//-----------------------------------------------------

// radial boundaries of each domain:
double r_limits[] = {0., 1., 2., __infinity} ;

Map_af map(mgrid, r_limits) ;   // Mapping construction

cout << map << endl ;

// Coordinates associated with the mapping:

const Coord& r = map.r ;
const Coord& x = map.x ;
const Coord& y = map.y ;
```



```
// Some scalar field to be used as a conformal factor
// --------------------------------------------------

Scalar psi4(map) ;

psi4 = 1 + 5*x*y*exp(-r*r) ;

psi4.set_outer_boundary(nz-1, 1.) ; // 1 at spatial infinity
                                    // (instead of NaN !)

psi4.std_spectral_base() ;  // Standard polynomial bases
                            // will be used to perform the
                            // spectral expansions
```

```
// Graphical outputs:
// ------------------

// 1D view via PGPLOT
des_profile(psi4, 0., 4., 1, M_PI/4, M_PI/4, "r", "\\gq\\u4") ;

// 2D view of the slice z=0 via PGPLOT
des_coupe_z(psi4, 0., -3., 3., -3., 3., "\\gq\\u4") ;

// 3D view of the same slice via OpenDX
psi4.visu_section('z', 0., -3., 3., -3., 3.) ;

cout << "Coefficients of the spectral expansion of Psi^4:"
     << endl ;
psi4.spectral_display() ;

arrete() ; // pause (waiting for return)
```



```
// Components of the flat metric in an orthonormal
// spherical frame :

Sym_tensor fij(map, COV, map.get_bvect_spher()) ;
fij.set(1,1) = 1 ;
fij.set(1,2) = 0 ;
fij.set(1,3) = 0 ;
fij.set(2,2) = 1 ;
fij.set(2,3) = 0 ;
fij.set(3,3) = 1 ;

fij.std_spectral_base() ;  // Standard polynomial bases will
                // be used to perform the spectral expansions

// Components of the physical metric in an orthonormal
// spherical frame :

Sym_tensor gij = psi4 * fij ;
```

```
// Construction of the metric from the covariant components:

Metric gam(gij) ;

// Construction of a Vector : V^i = D^i Psi^4 = (Psi^4)^{;i}

Vector vv = psi4.derive_con(gam) ;  // this is spherical comp.
                                    // (same triad as gam)

vv.dec_dzpuis(2) ; // the dzpuis flag (power of r in the CED)
                   // is set to 0 (= 2 - 2)

// Cartesian components of the vector :
Vector vv_cart = vv ;
vv_cart.change_triad( map.get_bvect_cart() ) ;

// Plot of the vector field :

des_coupe_vect_z(vv_cart, 0., -4., 1., -2., 2., -2., 2.,
                 "Vector V") ;
```



```
// A symmetric tensor of valence 2 : the Ricci tensor
//   associated with the metric gam :
//------------------------------------------------

Sym_tensor tens1 = gam.ricci() ;

const Sym_tensor& tens2 = gam.ricci() ; // same as before except
                        // that no memory is allocated for a
                        // new tensor: tens2 is merely a
                        // non-modifiable reference to the
                        // Ricci tensor of gam

// Plot of tens1

des_meridian(tens1, 0., 4.,"Ricci (x r\\u3\\d in last domain)",
             10) ;
```

```
// Another valence 2 tensor : the covariant derivative of V
//    with respect to the metric gam :
//-------------------------------------------------------
Tensor tens3 = vv.derive_cov(gam) ;

const Tensor& tens4 = vv.derive_cov(gam) ;

// the reference tens4 is preferable over the new object tens3
// if you do not intend to modify tens4 or vv, because it does
// not perform any memory allocation for a tensor.

// Raising an index with the metric gam :

Tensor tens5 = tens3.up(1, gam) ; // 1 = second index (index j
                // in the covariant derivative V^i_{;j})

Tensor diff1 = tens5 - vv.derive_con(gam) ; // this should be 0

// Check:
cout << "Maximum value of diff1 in each domain : " << endl ;
Tbl tdiff1 =  max(diff1) ;
```



```
// Another valence 2 tensor : the Lie derivative
// of R_{ij} along V :

Sym_tensor tens6 = tens1.derive_lie(vv) ;

// Contracting two tensors :

Tensor tens7 = contract(tens1, 1, tens5, 0) ; // contracting
                        // the last index of tens1 with the
                        // first one of tens5

// self contraction of a tensor :

Scalar scal1 = contract(tens3, 0, 1) ; // 0 = first index,
                                       // 1 = second index
```

```
// Each of these fields should be zero:

Scalar diff2 = scal1 - vv.divergence(gam) ; // divergence

Scalar diff3 = scal1 - tens3.trace() ;      // trace

// Check :
cout << "Maximum value of diff2 in each domain : "
    << max(abs(diff2)) << endl ;

cout << "Maximum value of diff3 in each domain : "
    << max(abs(diff3)) << endl ;

arrete() ;
```



```
// Tensorial product :

Tensor_sym tens8 = tens1 * tens3 ;  // tens1 = R_{ij}
                                    // tens3 = V^k_{;l}
                                    // tens8
                                    //   = (T8)_{ij}^k_l
                                    //   = R_ij V^k_{;l}

cout << "Valence of tens8 : " << tens8.get_valence()
     << endl ;

cout <<
"Spectral coefficients of the component (2,3,1,1) of tens8:"
     << endl ;

tens8(2,3,1,1).spectral_display() ;
```

```
    ////////////////////////////////////////////////////
    //                                                //
    // To see more functions, please have a look to   //
    // Lorene documentation at                        //
    // http://www.lorene.obspm.fr/Refguide/           //
    //                                                //
    ////////////////////////////////////////////////////

    return EXIT_SUCCESS ;

}
```

I. FIELD MANIPULATION WITH LORENE

The aim is simply to get used to LORENE library for the definition, manipulation, computation and drawing of scalar and vector fields in spherical coordinates and/or components. For all classes and functions, please look carefully at the documentation at `Lorene/Doc/refguide/index.html`.

- Setup a multi-domain three-dimensional grid. It should contain a nucleus, one or more shells and a compactified external domain. Take it to be symmetric / equatorial plane and not symmetric / $(x, y) \to (-x, -y)$.
- Using coordinate fields (`Coord` objects, members of the mapping), define a *regular* 3D (but symmetric / equatorial plane) scalar field of type `Scalar`.
- After setting the spectral base, draw iso-contours with `des_coupe_...` and profiles with `des_meridian`.
- Compute the radial derivative of the field and compare it to the "analytic" value (*e.g.* using `maxabs` or `diffrelmax`).
- Define a regular vector field in spherical triad, either by setting it first in a Cartesian triad and changing the triad, or as the gradient of a scalar field (covariant derivative / flat metric). Draw the vector field.

II. TEST OF A ROTATING BLACK HOLE METRIC

With tensor calculus tools, it is easy to check whether a given metric is solution of Einstein equations. As an example, the Kerr-Schild metric shall be tested, within the framework of the 3+1 formalism. This metric provides a description of a rotating black hole (*i.e.* vacuum space-time), with a mass M and the angular momentum per unit mass a:

$$g_{\mu\nu} = f_{\mu\nu} + 2H l_\mu l_\nu; \tag{1}$$

where $f_{\mu\nu}$ is the flat metric,

$$H = \frac{M\rho^3}{\rho^4 + a^2 z^2} \tag{2}$$

and

$$l_\mu = \left(1, \frac{\rho x + a y}{\rho^2 + a^2}, \frac{\rho y - a x}{\rho^2 + a^2}, \frac{z}{\rho}\right). \tag{3}$$

Note that the spatial components of l_μ are expressed in a Cartesian triad and ρ is related to the usual radial coordinate r – (x, y, z) being the usual Cartesian coordinates – by the relation:

$$\rho^2 = \frac{1}{2}\left(r^2 - a^2\right) + \sqrt{\frac{1}{4}\left(r^2 - a^2\right)^2 + a^2 z^2}. \tag{4}$$

To test it, the metric should be written in the 3+1 form[1] (using only 3-tensors):

$$g_{\mu\nu}\, dx^\mu\, dx^\nu = -N^2 dt^2 + \gamma_{ij}(dx^i + \beta^i dt)(dx^j + \beta^j dt); \tag{5}$$

with N being the lapse, β the shift and γ_{ij} the 3-metric. In this case:

$$\begin{aligned} N &= \frac{1}{\sqrt{1+2H}}; \\ \beta_i &= 2H l_i; \\ \gamma_{ij} &= f_{ij} + 2H l_i l_j \end{aligned}$$

[1] latin indices range from 1 to 3 (only spatial components), whereas greek ones range from 0 to 3

One also defines the extrinsic curvature

$$K_{ij} = \frac{1}{2N}\left(\pounds_{\boldsymbol{\beta}}\gamma_{ij} - \frac{\partial}{\partial t}\gamma_{ij}\right) \tag{6}$$

$\pounds_{\boldsymbol{\beta}}\gamma_{ij}$ being the Lie-derivative along the shift of the 3-metric.

The ten Einstein equations write (in vacuum):

- the Hamiltonian constraint equation:

$$R + K^2 - K_{ij}K^{ij} = 0, \tag{7}$$

- the three momentum constraint equations

$$D_j K_i{}^j - D_i K = 0, \tag{8}$$

- and the six dynamical evolution equations

$$\frac{\partial}{\partial t}K_{ij} - \pounds_{\boldsymbol{\beta}}K_{ij} = -D_i D_j N + N\big[R_{ij} - 2K_{ik}K^k_{\ j} + KK_{ij}\big]. \tag{9}$$

D_i is the covariant derivative / γ_{ij}, K the trace of K_{ij}, R_{ij} and R the Ricci tensor and scalar associated with this 3-metric.

III. SUGGESTED STEPS

- Define a grid (symmetric / $(x, y) \to (-x, -y)$ transform), with at least 4 points in φ to be able to rotate from Cartesian triad to the spherical one. Either this grid is without the nucleus, to excise the black hole singularity, or all fields should be set to 0 or 1 in the nucleus to discard the divergence near the centre. Define a mapping on that grid.
- Setup the Kerr-Schild metric described above, with the lapse, shift and the 3-metric.
- Verify that the 1+3+6 equations above are satisfied, using the appropriate methods of classes `Tensor`, `Vector` and `Metric`. Be very careful with the `dzpuis` flag!

System of equations. Application to Yang-Mills-Higgs monopole

Philippe Grandclément

Laboratoire de l'Univers et de ses Théories (LUTH)
CNRS / Observatoire de Paris
F-92195 Meudon, France

philippe.grandclement@obspm.fr

Collaborators
Silvano Bonazzola, Eric Gourgoulhon, Jérôme Novak

November 14-18, 2005

Outline

INTRODUCTION

Type of problems

We have to solve a set of k partial differential equations, coupled :

$$H_i f_i = S_i\left(f_1, f_2 .... f_k\right) \quad \forall 0 \leq i < k$$

where H_i are differential operators (typically second order...)

Iteration technique

- Give an initial guess for the f_i.
- Computes the sources.
- Invert the operators H_i.
- If the relative change in the f_i is small stop, else compute the new sources and loop.

A few questions

Choice of H_i and f_i

- The choice of operators and functions can greatly influence the stability and convergence of the code.
- Typically, it is best if S_i contains only quadratic terms (or even higher order) in f_k.

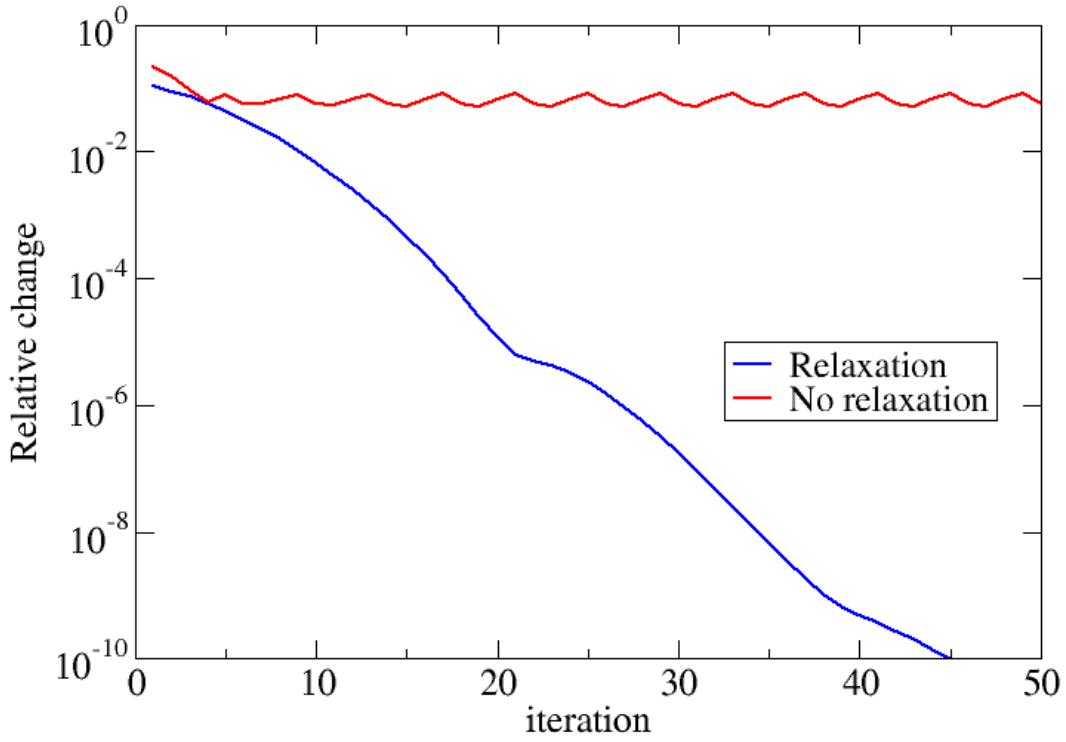
Relaxation

- If we replace simple f_i par $H_i^{-1}[S_i]$ the code usually diverges.
- We slow the change from step to step by using relaxation like :

$$f_i^{\text{new}} = \lambda H_i^{-1}[S_i] + (1-\lambda) f_i^{\text{old}}$$

- Typical values : $\lambda \approx 0.5$.

Influence of relaxation



H_i : simple cases

In LORENE a lot of choices are implemented for the operators H_i.

Members of `Scalar` :

- First order : primitive `primr`.
- Standard Poisson : `poisson` and `poisson_tau`.
- Poisson with inner boundary conditions : `poisson_dirichlet`, `poisson_neumann`, `poisson_dir_neum`.
- Angular part : `poisson_angu`.

The vectorial counterpart do also exist, members of `Vector`.
However, we may want to change H_i and f_i from domains to domains : need a more general solver...

YANG-MILLS-HIGGS MONOPOLE

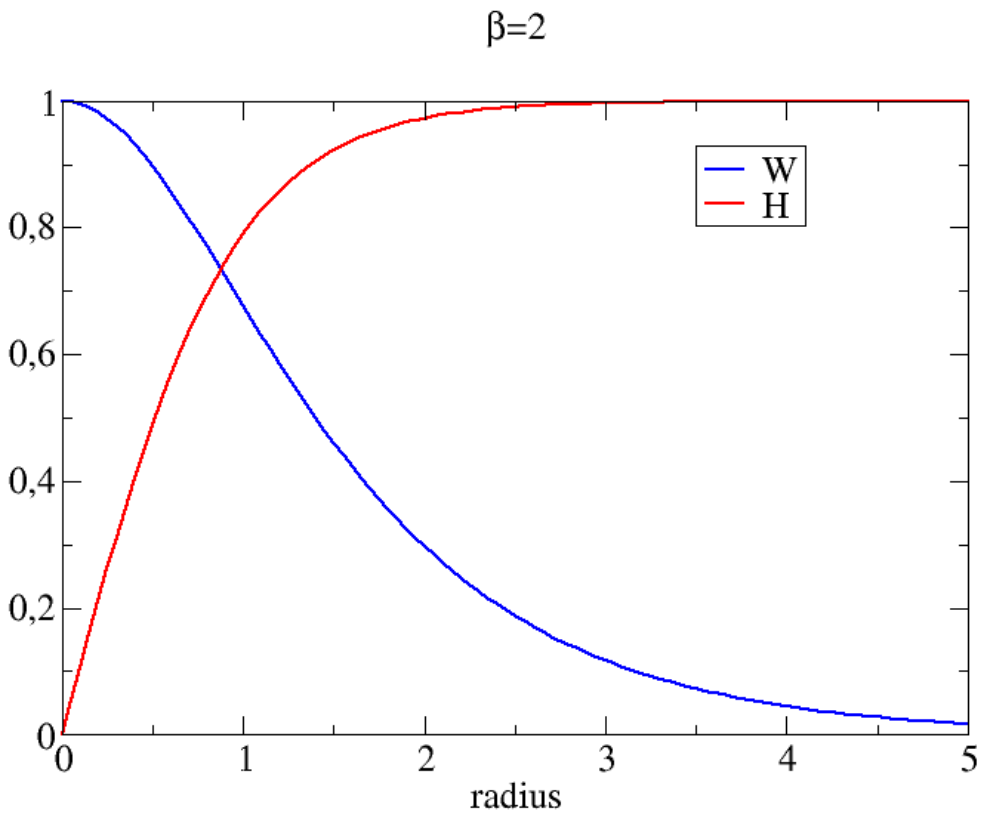
The equations

Monopoles are stationary, localized solutions of finite energy, in certain theories. Such solutions, in Yang-Mills-Higgs, assuming spherical symmetry, are solutions of a set of two equations :

$$\begin{aligned} W'' &= \frac{W\left(W^2-1\right)}{r^2} + WH^2 \\ H'' + \frac{2}{r}H' &= 2\frac{W^2 H}{r^2} + \frac{\beta^2}{2}H\left(H^2-1\right) \end{aligned}$$

where W describe the gauge field, H the Higgs field, and β is a parameter giving the ratio of the masses.

Typical solutions



Asymptotic behavior

Near the origin :

- $W = 1 - ar^2 + \mathcal{O}\left(r^4\right)$
- $H = br + \mathcal{O}\left(r^3\right)$

At infinity

- W goes to zero exponentially.
- H goes to 1 exponentially.

Why not use Poisson operators ?

Suppose we write the system in the following form :

$$\begin{aligned} \Delta W &= S_W\left(W, H\right) \\ \Delta H &= S_H\left(W, H\right) \end{aligned}$$

- If we do not start from THE solution, after the first step, homogeneous solutions of the Laplacian do appear.
- Those homogeneous solutions are in $1/r$ and do not decay fast enough for the sources.
- the code crashed very quickly.

One needs to maintain exponential convergence throughout the iteration.

Equations near infinity

In the CED, one will work with W and $h = H - 1$.
One can make Helmholtz operators appear :

$$\begin{aligned} \Delta_{l=0} W - W &= hW(h+2) + \frac{W\left(W^2-1\right)}{r^2} + 2\frac{W'}{r} \\ \Delta_{l=0} h - \beta^2 h &= 2\frac{W^2(h+1)}{r^2} + \frac{\beta^2}{2} h^2 (h+3) \end{aligned}$$

Homogeneous solutions are exponentials $\Longrightarrow$ exponential convergence is maintained during the iteration.

Near the origin

- One can see that $W = 0$ everywhere is solution of the system of equations.
- One should prevent the code to converge to $W = 0$.
- Near the origin, one can use :

$$w = \frac{W-1}{r}$$

Given that : $W = 1 - ar^2 + \mathcal{O}\left(r^4\right)$

- w is regular at the origin.
- w is odd near the origin.
- It forces $W(r=0) = 1$.

Equations near the origin

In the nucleus, one will work with w and H.
The equations can be put in the following form :

$$\begin{aligned}\Delta_{l=1} w \equiv w'' + \frac{2}{r} w' - 2\frac{w}{r^2} &= w^3 + 3\frac{w^2}{r} + (1 + rw)\frac{H^2}{r} \\ \Delta_{l=1} H \equiv H'' + \frac{2}{r} H' - 2\frac{H}{r^2} &= 2H\left(w^2 + 2\frac{w}{r}\right) + \frac{\beta^2}{2} H\left(H^2 - 1\right).\end{aligned}$$

Equations in the shells

Neither of the functions are singular.
We can use W and H and write the equations like :

$$\begin{aligned}\Delta_{l=0} W - W &= W\left(H^2 - 1\right) + \frac{W\left(W^2 - 1\right)}{r^2} + 2\frac{W'}{r} \\ \Delta_{l=0} H - \beta^2 H &= 2\frac{W^2 H}{r^2} + \frac{\beta^2}{2} H\left(H^2 - 3\right)\end{aligned}$$

THE Param_elliptic CLASS

So what is needed ?

We need to solve equations of the type : $H_d f_d = S_d$ where d denotes the domain

- H_d can be different operators in each domain.
- f_d are auxiliary variables, related to the "real" one F.
- The auxiliary variables can be different from domain to domain.
- The continuous function is F.

Use the Param_elliptic object

- constructor : `Param_elliptic(const Scalar & so)`
- various functions to initialize the `Param_elliptic`, setting both
  - the H_d
  - the variable changes.
- Solve the equation by calling `Scalar::sol_elliptic (Param_elliptic)`
- It returns a `Scalar` containing f_d.

Setting the variable change

For now the auxiliary variables should be of the type :

$$W = F(r, \theta, \varphi) + G(r)\, w$$

F and G are given in every domain

- By default $F = 0$ and $G = 1$
- F is changed by `Param_elliptic::set_variable_F (const Scalar &)`
- G is changed by `Param_elliptic::set_variable_G (const Scalar &)`.
- Be careful : the code does not check that G is only a function of r

Monopole variable change

For W

- In the nucleus : $W = 1 + rw$
- W elsewhere.

For H

- In the CED : $H = 1 + h$
- H elsewhere.

They are of the right type

Changing the operators

Use the provided member functions of `Param_elliptic` like :

- `set_helmholtz_minus (int zone, double m, Scalar &so)`

  sets the operator to $\Delta - m^2$ in the domain `zone`.

- `inc_l_quant (int zone)`

  increases l in the domain `zone`, the operator being of the type :

$$f'' + \frac{2}{r} f' - \frac{l\,(l+1)}{r^2} f$$

I. THE PROBLEM

We propose to solve the system for a static spherically symmetric Yang-Mills-Higgs monopole. Using the minimal spherically symmetric Ansatz, the solution is described by two functions : one describing the gauge field W and one the Higgs field H. Those two functions depend only on r and obey a system of two coupled equations :

$$W'' = \frac{W\left(W^2-1\right)}{r^2} + WH^2 \tag{1}$$

$$H'' + \frac{2}{r}H' = 2\frac{W^2H}{r^2} + \frac{\beta^2}{2}H\left(H^2-1\right) \tag{2}$$

The only parameter of the solution is β constraining the mass of the Higgs field. We will restrict ourselves to the cases $0 < \beta < \infty$.

II. ASYMPTOTIC BEHAVIORS

Near the origin, one has the following behaviors :

$$W = 1 - ar^2 + \mathcal{O}\left(r^4\right) \tag{3}$$

$$H = br + \mathcal{O}\left(r^3\right) \tag{4}$$

At infinity, the fields converge exponentially, i.e. W and $h = H - 1$ go to zero exponentially.

III. SYSTEM IN VARIOUS DOMAINS

- **In the nucleus** : one uses $w = \dfrac{W-1}{r}$ and H (odd functions near the origin) and rewrite the system as :

$$\Delta_{l=1}w \equiv w'' + \frac{2}{r}w' - 2\frac{w}{r^2} = w^3 + 3\frac{w^2}{r} + (1+rw)\frac{H^2}{r} \tag{5}$$

$$\Delta_{l=1}H \equiv H'' + \frac{2}{r}H' - 2\frac{H}{r^2} = 2H\left(w^2 + 2\frac{w}{r}\right) + \frac{\beta^2}{2}H\left(H^2-1\right). \tag{6}$$

- **In the shells** : one uses W and H but rewrites the system to make the Helmholtz operators appear (optional) :

$$\Delta_{l=0}W - W = W\left(H^2-1\right) + \frac{W\left(W^2-1\right)}{r^2} + 2\frac{W'}{r} \tag{7}$$

$$\Delta_{l=0}H - \beta^2 H = 2\frac{W^2H}{r^2} + \frac{\beta^2}{2}H\left(H^2-3\right) \tag{8}$$

- **In the external domain** : one works with W and $h = H - 1$ and make Helmholtz operators appear :

$$\Delta_{l=0}W - W = hW(h+2) + \frac{W\left(W^2-1\right)}{r^2} + 2\frac{W'}{r} \tag{9}$$

$$\Delta_{l=0}h - \beta^2 h = 2\frac{W^2(h+1)}{r^2} + \frac{\beta^2}{2}h^2(h+3) \tag{10}$$

IV. SUGGESTED STEPS

- Look at the proposed `Monopole` class that contains W, H, w and h (each of them being a `Scalar`).
- Implement functions that initialize W and H, with the right behaviors and basis. Plot the results.
- Implement functions that go from W to w and from H to h and conversely. Plot the various functions.

- Compute the sources in various domains and plot them.
- Setup the main iteration loop, based on `Param_elliptic` class.
- For various moderate values of β, compute a and b appearing in Eqs. (3) and (4).
- Try to go to high values of β.

V. SOLVING THE SYSTEM ON TWO GRIDS

For high values of β, one can show that H varies on a relative length scale $\propto 1/\beta$ whereas W varies always on length of the order unity. So, for high values of β, those two functions vary on very different length scales, causing the code to crash. To cope with that, one can use two grids :

- one on scales of the order 1, used to solve the equation for W.
- one on scales of the order $1/\beta$, used to solve the equation for H

This can be implemented by describing all the fields (W, w, H and h) on two sets of grids. One can go from one grid to the other by using the `Scalar::import()` function. Be careful : this should only be used with continuous functions, to avoid Gibbs phenomenon. Verify that the use of two grids enables to go to very high values of β.

Singular elliptical operators

..... by

——— *S. BONAZZOLA*

L.U.T.H Oservatoire Paris Meudon

In problems involving one (or more Black Holes (B.H.) when the excision technique is used, we can have to handle degenerate elliptical operators.
An example, is the equation for the shift β^i when the lapse N vanishes on the horizon. In fact the equation for the shift reads (in an apropiate gauge)

$$\nabla^j K_{ij} = 0 \qquad (1)$$

where K^{ij} is the extrinsic curvature tensor

$$K_{ij} = \frac{1}{2N}(\nabla_i \beta_j + \nabla_j \beta_i - \partial_0 \gamma_{ij}) \qquad (2)$$

Here in after we shall express all the differential operators in terms of the flat covariant derivative $\mathcal{D}_i$ computed with respect the flat metric f_{ik} that in spherical coordinates reads [1]

$$f_{11} = 1, \quad f_{22} = r^2, \quad f_{33} = r^2 \sin^2 \theta \tag{3}$$

Under the hypothesis that the topology of the horizon is the topology of the sphere the equation of the horizon can be reduced to be

$$r = 1 \tag{4}$$

Consequently we have to solve the the Einstein equations in the *excised* space

$$1 \leq r \leq \infty \tag{5}$$

The technique used to solve the Einstein equations is to solve these equations in two domains

$$1 \leq r \leq 2, \qquad 2 \leq r \leq \infty \tag{6}$$

and to match the solutions and they first derivatives at $r = 2$
The shift equation(1) can be written

[1] See the paper by S.Bonazzola et al. *Phys.Rev.D* **70** (2004), 104007

$$\mathcal{D}_j\mathcal{D}^j\beta^i+\frac{1}{3}\mathcal{D}^i(\mathcal{D}^j\beta^j)-(\mathcal{D}^i\beta^j+\mathcal{D}^j\beta^i-\frac{2}{3}\mathcal{D}_l\beta^l f^{ij}+S_1^i)\frac{\partial_j N}{N}=S_2^i$$

With the B.C. $\beta^i=0\mid_{r=\infty}$. In order to match the solution and its derivative we must have at list one homogeneous solution in the domain $1\leq r\leq 2$. Question : How many homogeneous solutions exist ?
Taking into account that near the horizon

$$N=(r-1)N_0(r,\theta,\phi)$$

we look for the homogeneous solutions of the equation

$$\mathcal{D}_j\mathcal{D}^j\beta^i-\frac{1}{3}\mathcal{D}^i\mathcal{D}_j\beta^i-\frac{1}{x}(\mathcal{D}^r\beta^i+\mathcal{D}^i\beta^r-\frac{2}{3}\mathcal{D}_l\beta^l f^{ir})=0$$

where

$$x=r-1$$

The vectorial operator of the above equation, in spherical coordinates and spherical components

is quite messy. By introducing two angular potentials η and μ defined by the equations

$$\beta^\theta = \partial_\theta \eta - \frac{1}{\sin\theta}\partial_\phi \mu, \quad \beta^\phi = \partial_\theta \mu + \frac{1}{\sin\theta}\partial_\phi \eta$$

we have two coupled Poisson equations for β^r and η and a Poisson equation for μ that after an expansion in spherical harmonics reads:

$$\frac{d^2\mu}{dr^2} + \frac{2}{r}\frac{d\mu}{dr} - \frac{l(l+1)}{r^2}\mu - \frac{1}{x}\left(\frac{d\mu}{dr} - \frac{\mu}{r}\right) = 0, \quad (x = r - 1)$$

A solution μ_1 can be found by making a power expansion

$$\mu_1 = x^2 - \frac{5}{3}x^3 + ...$$

For $l = 1$ it exists an other homogeneous solution:

$$\mu_2 = r$$

that means that a black hole can rigidly rotate. In fact, the non vanishing at $r = 1$ hmogenous

solutions are

$$\mu_1 = r\cos\theta, \quad \mu_2 = r\sin\theta\cos\phi, \quad \mu_3 = r\sin\theta\sin\phi \tag{7}$$

from wich the corresponding solutions for β

$$\beta_r = 0, \quad \beta_\theta = 0, \quad \beta_\phi = r\sin\theta$$

$$\beta_r = 0, \quad \beta_\theta = r\sin\phi, \quad \beta_\phi = r\cos\theta\cos\phi$$

$$\beta_r = 0, \quad \beta_\theta = -r\cos\phi, \quad \beta_\phi = r\cos\theta\sin\phi$$

A similar analysis can be performed for the poloydal part β^r, η of th shift. The conclusions are:

For $l = 1$, two couples of homogeneous solutions exist. That means that a rigid translation of the horizon can be chosen.

For $l \neq 1$ only β^r can be given on the horizon: The horizon can breath.

Finally singular equations exist for the metric coefficients h^{ik}. For some coefficient (h^{rr}) a

boundary condition on the horizon can be given, for other coefficients ($h^{\theta\theta}$) not.

Equation $\triangle G + \frac{1}{r-1}(k_1\frac{d}{dr} + \frac{k_2}{r})G = 0$

Consider the equation

$$\frac{d^2G}{dr^2}+k_0\frac{1}{r}\frac{dG}{dr}+\frac{1}{r^2}(-l(l+1)+k_l)G+\frac{1}{r-1}(k_1\frac{d}{dr}+\frac{k_2}{r})G=0 \tag{8}$$

For $k_1 = k_2 = 0$ the above equation has two regular solution regular a t $r = 0$ and at $r = \infty$,k_l if $k_l = ((k_0 - 1)^2 - 1)/4$.

$$g_1 = r^j, j_1 = \frac{1 - k_0 - (2l + 1)}{2}, j_2 = \frac{1 - k_0 + (2l + 1)}{2}$$

Note that the two solutions r^{j_1} and r^{j_2} are integer numbers if k_0 is integer to. In this section, we study the number and the aanalytical properties of the solution for different values of the parameter k_0, k_1, k_2.

Case $k_1 \neq 0$ **and** $k_1 \neq \mid 1 \mid$

Without losses of generality we consider only the case $k_2 = 0$. In fact by putting $\bar{G} = Gr^{k_2/k1}$

the equation for the new function $\bar{G}$ will be transformed in an equation having $k_2 = 0$
The case $\mid k_1 \mid = 1$ was already discussed.

The technique used consists in studying the behavior of the solution around the singular point $r = 1$. For that we introduce the new variable $x = r - 1$. The Eq. 8 writes

$$\frac{d^2G}{dx^2} + k_0\frac{dG}{dx} + (-l(l+1) + k_l)G + \frac{1}{x}k_1\frac{d}{dx}G = 0 \tag{9}$$

We look for an homogeneous solution $H_1(x)$ by making a series expansion

$$H_1(x) = a_0 + a_2x^2 + a_3x^3 + ...$$

. The coefficients a_0 and a_2 must satisfy the relation

$$2(1 + k_1)a_2 + (-l(l + 1) + k_l)a_0 = 0 \tag{10}$$

we see that $k_1 = -1$ Tthe pathological case $a_0 = 0$ and the nonvanishing homogeneous solution does not exist.

A second homogeneous solution $H_2(x)$ can be found by searching a solution that vanishes at $x = 0$, $(r = 1)$. We put $H_2(x) = x^j$ we obtain

$$j(j-1) + jk_1 = 0$$

from which

$$j = -k_1 + 1 \tag{11}$$

Therefore $H_2(x)$ will be

$$H_2(x) = x^j(1 + a_{j+1}x + ...) \tag{12}$$

where j is given by the Eq.(11) we see that if $k_1 < 1$ then the solution is regular, moreover if k_1 is integer number $k_1 \leq -2$ the solution has a polynomial behavior near the singularity.
Conclusions: If $k_1 < 2$ Then it exist two independent homogeneous solutions of the equation Eq.8

Numerical solution of the homogeneous equations

If a non vanishing solution exists we shall proceed in the following way take a solution of the first order differential equation appearing in the singular term of the Eq.(8):

$$g_0 = r^{\frac{-k_2}{k_1}} \tag{13}$$

This solution, in general is not a solution of the the second order equation (8) Introduce g_0 in the Eq.(8) and compute the rest R. Solve the non homogeneous equation

$$\frac{d^2G}{dr^2}+\frac{k_0}{r}\frac{dG}{dr}+\frac{1}{r^2}(k_l-l(l+1))G+\frac{1}{x}(k_1\frac{d}{dr}+\frac{k_2}{r})G = -R \tag{14}$$

with the Galerkin approximation by using a new set of function Φ_n vanishing as x^2. We can use the set of (non orthogonal functions)

$$\Phi_n = (r-1)^2 T_n(r) \tag{15}$$

Let be g_p this particular solution, The homoge-

neous H_1 solution of the EQ.(8) will be

$$H_1 = g_p + g_0 \tag{16}$$

Numerical implementation

In this section I will show how to find numerically the homogeneouwe see that if $k_1 < 1$ then the solution is regular, moreover if k_1 is integer number $k_1 \leq -2$ the solution has a polynomial behavior near the singularity.
Conclusions: If $k_1 < 2$ Then it exist two independent homogeneous solutions of the equation Eq.8
Numerical solution of the homogeneous equations

If a non vanishing solution exists we shall proceed in the following way take a solution of the first order differential equation appearing in the singular term of the Eq.(8):

$$g_0 = r^{\frac{-k_2}{k_1}} \tag{17}$$

This solution, in general is not a solution of the the second order equation (8) Introduce g_0 in the Eq.(8) and compute the rest R. Solve the non homogeneous equation

$$\frac{d^2G}{dr^2}+\frac{k_0}{r}\frac{dG}{dr}+\frac{1}{r^2}(k_l-l(l+1))G+\frac{1}{x}(k_1\frac{d}{dr}+\frac{k_2}{r})G=-R \tag{18}$$

with the Galerkin approximation by using a new set of function Φ_n vanishing as x^2. We can use the set of (non orthogonal functions)

$$\Phi_n=(r-1)^2T_n(r) \tag{19}$$

Let be g_p this particular solution, The homogeneous H_1 solution of the EQ.(refeqg) will be

$$H_1=g_p+g_0 \tag{20}$$

Numerical implementation

In this section I will show how to find numerically the hogeneos solutions. We shall consider

the solution H_2 that vanishes at $r = 1$

Let be $\mathcal{O}_i^j$ the matrix of the operator of the equation Eq.(9)

$$\mathcal{O} = r^2\frac{d^2}{dr^2} + r\frac{d}{dr} + k_l - l(l+1) + \frac{r}{x}(rk_1\frac{d}{dr} + k_2) \tag{21}$$

with respect the Galerkin basis

$$\Phi_n(r) = (r-1)^2 T_n(r)$$

Finding H_2 it means to find the coefficients a_n of the expansion

$$H_2(r) = \Sigma\, a_n \Phi_n(r)$$

Consequently we have to find a non trivial solution of the algebraic system of equations

$$\mathcal{O}_i^j\, a_j = 0 \tag{22}$$

A such a solution exists because the determinant of the matrix $\mathcal{O}_i^j$ vanishes. We shall replace the last line of the system (22) by

$$\mathcal{O}_N^j = 1, 0, 0, ...$$

and we impose that the first coefficient a_1=1 the system will look as

$$\mathcal{O}_1^1\, a_1 + \mathcal{O}_1^2\, a_2 + \mathcal{O}_1^3\, a_3 + ... = 0$$
$$\mathcal{O}_2^1\, a_1 + \mathcal{O}_2^2\, a_2 + \mathcal{O}_2^3\, a_3 + ... = 0$$
$$..... = 0$$
$$a_1 + 0 + 0 + 0 + 0 + 0 + ... = 1$$

Solution of the inhomogeneos equations
(The pathological case)

We shall consider the solution of the thoroidal component of the shift:

$$r^2\frac{d^2\mu}{dr^2}+2r\frac{d\mu}{dr}-l(l+1)\mu+\frac{r^2}{x}(-\frac{d\mu}{dr}+\frac{\mu}{r}+S_1) = r^2 S_2 \tag{23}$$

($x = r - 1$) The case $k_1 = -1$ is pathological, because it exists a non vanishing homogeneous soluttion at $r = 1$ only for $l = 1$. In order to handle the singular term S_1/x we define a new

function

$$\tilde{S}_1 = S_1(r) - q$$

where $q = S_1(1)$. Thus function vanishes at $r = 1$ an we re=writw the above equation as

$$r^2\frac{d^2\mu}{dr^2}+2r\frac{d\mu}{dr}-l(l+1)\mu+\frac{r^2}{x}(-\frac{d\mu}{dr}+\frac{\mu}{r}+q) = r^2(S_2+\frac{\tilde{S}_1}{x}) \tag{24}$$

We look for a solution $\tilde{\mu}$ such thta

$$\tilde{\mu} = -qx + F(r)$$

where Fr vanishes as x^2 at $r = 1$. By replacing $\tilde{\mu}$ we have

$$r^2\frac{d^2F}{dr^2}+2r\frac{dF}{dr}-l(l+1)F+\frac{r^2}{x}(-\frac{dF}{dr}+\frac{F}{r}) \tag{25}$$

$$= r^2[S_2 + \frac{\tilde{S}_1}{x} - q(3r - l(l+1)x] \tag{26}$$

and the solution is obtained by expanding F on the Galerkin base as was done beore.

and the solution is obtained by expanding F on the Galerkin base as was done before. [2]

[2]Note that if a regular solution is required, the source must vanishes at$r = 1$

Finding Kerr solution startinf from nothing

We show how construct an apprxomitatd Kerr solution as an application of the above formalism.

Start frm the flat metric f_{ik}.
First step:
Find a solution of the lapse equation

$$\triangle N = 0$$

with the B,C. $N(1) = 0$, $N(\infty) = 1$ This solution can be

$$N(r) = 1 - \frac{1}{r}$$

Second step:
Find a solution of the linearised eequation for Ψ^4

$$\triangle \Psi^4 = 0$$

where Ψ is the conformal factor. The solution

must satisfy the B.C. of an apparent horizon

$$\frac{d\Psi^4}{dr} = -1 \mid_{r=1}, \quad \Psi^4 = 1 \mid_{r=\infty}$$

This solution is

$$\Psi^4 = \frac{1}{r}$$

3^{th} step: Find (numerically) a solution for μ_l with the B.C.

$$\mu_l = \delta_l^1 \mu_0 \mid_{r=1}, \qquad \mu_l = 0 \mid_{r=\infty}$$

where δ_l^i is the Kronler δ Iterate
Note that th source of μ vanishes at $r = 1$ at each iterartion. (See footnote)

Fig. 1) shows the lapse N. Fig.2) shows the fonction $N_0 = N/x$ (in the first domain) Fig. 3) the shift (for different values of θ
The other figures show the convergence of the iteration.

Conclusion
We have studied the analytical properties of the

solutions of singular elliptical P.D.E. Spectral methods allows to us to compute numerical solutions of singular equations.
As examplese computed the Kerr solution within the conformally flat approximation. The algorithm has shown to be robust (in the sense that it converges exponentially without a relaxation parameter).

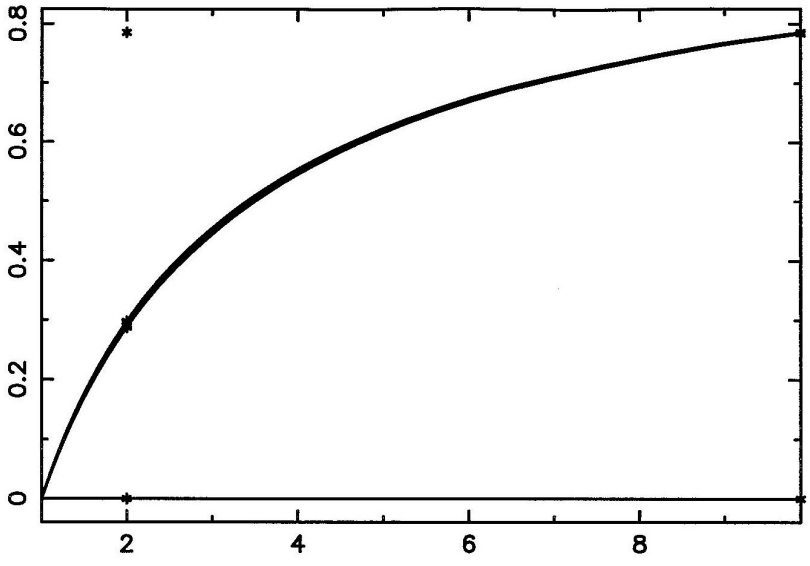


Fig. 1

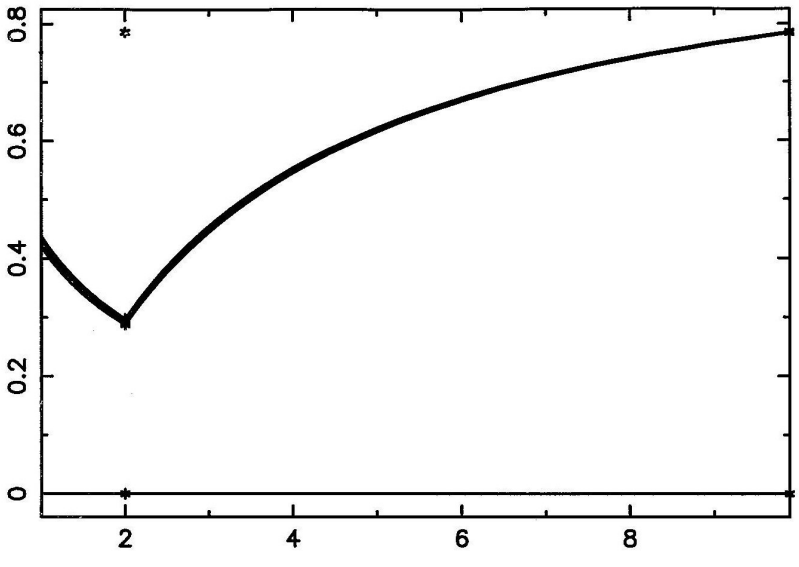


Fig. 2

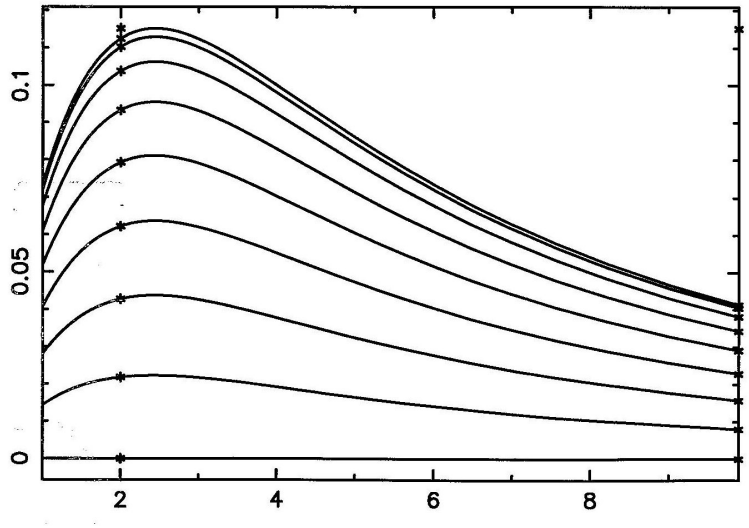


Fig. 3

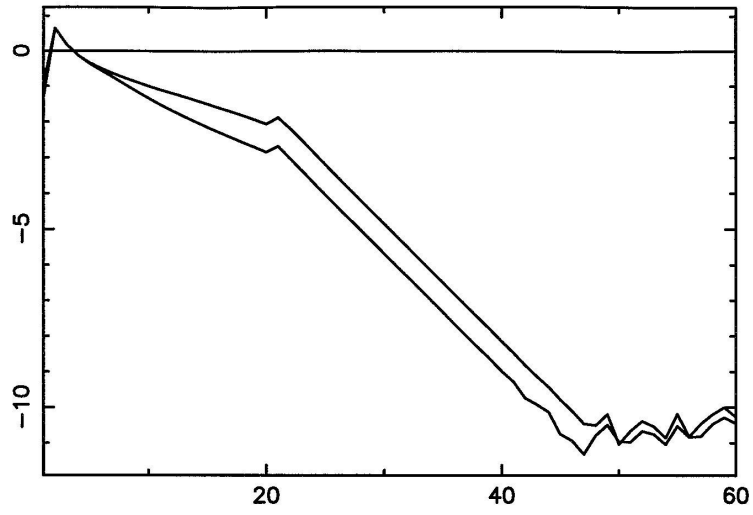


Fig. 4

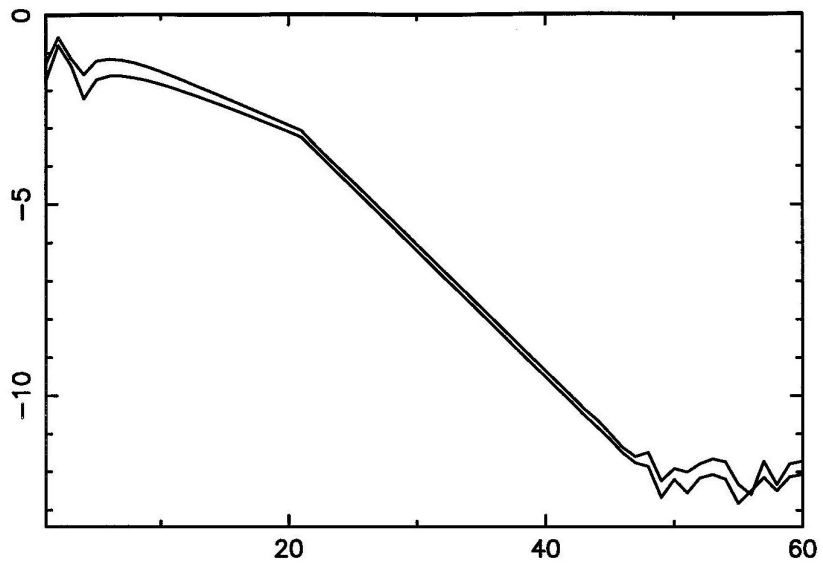


Fig. 5

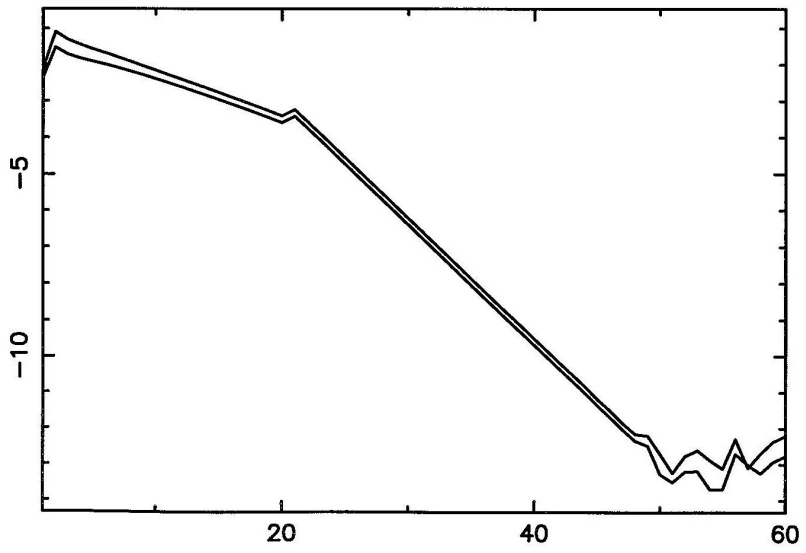


Fig. 6

Evolution equations with spectral methods: the case of the wave equation

Jérôme Novak

Jerome.Novak@obspm.fr

Laboratoire de l'Univers et de ses Théories (LUTH)
CNRS / Observatoire de Paris, France

in collaboration with
Silvano Bonazzola

November, 18 2005

Wave equation
Jérôme Novak
Time evolution
Time discretization
Integration schemes
Integration schemes
Wave equation
Explicit scheme
Implicit scheme
Boundaries
Outgoing conditions
Sommerfeld BC
Asymptotics
Enhanced BC

Plan

Time discretization

It seems that, in general, there is no efficient spectral decomposition for the time coordinate...
⇒use of finite-differences schemes ! t is discretized (usually) on an equally-spaced grid, with a times-step δt : $U^J = U(J \times \delta t)$.

$$\frac{dU}{dt} = F(U) = L(U) + Q(U)$$

Study, for different integration schemes of :

- stability : $\forall n \|U^n\| \leq Ce^{Kt}|U^0\|$, for some $\delta t < \delta_{\text{lim}}$,
- region of absolute stability : when considering

$$\frac{dU}{dt} = \lambda U,$$

the region in the complex plane for $\lambda\delta t$ for which $\|U^n\|$is bounded for all n,

- unconditional stability : if δ is independent from N (level of spectral truncation).

Wave equation
Jérôme Novak
Time evolution
Time discretization
Integration schemes
Integration schemes
Wave equation
Explicit scheme
Implicit scheme
Boundaries
Outgoing conditions
Sommerfeld BC
Asymptotics
Enhanced BC

One-dimensional study

To use the knowledge of the region of absolute stability, it is necessary to diagonalize the matrix L and study its eigen-values λ_i.
In one dimension :

First-order Fourier

For $L = d/dx$, one finds $\max|\lambda_i| = O(N)$

First-order Chebyshev

For $L = d/dx$, one finds $\max|\lambda_i| = O(N^2)$

Second-order Fourier

For $L = d/dx^2$, one finds $\max|\lambda_i| = O(N^2)$

Second-order Chebyshev

For $L = d^2/dx^2$, one finds $\max|\lambda_i| = O(N^4)$

Time integration schemes
Most popular...

Explicit

- first-order Adams-Bashford scheme (a.k.a forward Euler) :

$$U^{n+1} = U^n + \delta t F\left(U^n\right),$$

- second-order Adams-Bashford scheme :

$$U^{n+1} = U^n + \delta t \left[\frac{23}{12} F\left(U^n\right) - \frac{16}{12} F\left(U^{n-1}\right) + \frac{5}{12} F\left(U^{n-2}\right)\right],$$

- Runge-Kutta schemes...

All these exhibit a bounded region of absolute stability
$\Rightarrow \exists K > 0, \quad \delta t \leq K / \max|\lambda_i|$ (Courant condition ...).

Observatoire de Paris LUTH
Laboratoire de l'Univers et de ses Théories
Wave equation
Jérôme Novak
Time evolution
Time discretization
Integration schemes
Integration schemes
Wave equation
Explicit scheme
Implicit scheme
Boundaries
Outgoing conditions
Sommerfeld BC
Asymptotics
Enhanced BC

Time integration schemes
Most popular...

Implicit

Adams-Moulton :

- first-order (a.k.a backward Euler scheme)

$$U^{n+1} = U^n + \delta t F\left(U^{n+1}\right),$$

- second-order (a.k.a. Crank-Nicholson scheme)

$$U^{n+1} = U^n + \frac{1}{2}\delta t \left[F\left(U^{n+1}\right) + F\left(U^n\right)\right].$$

Both have an unbounded region of absolute stability in the left complex half-plane $\Rightarrow$unconditionally stable schemes.
Higher-order AM schemes have only a bounded region of absolute stability.

Schemes can be mixed and various source terms can be treated in different ways (*e.g.* linear $\Rightarrow$implicit / non-linear $\Rightarrow$explicit).

Wave equation

The three-dimensional wave equation in spherical coordinates :

$$\Box\phi = -\frac{1}{c^2}\frac{\partial^2\phi}{\partial t^2} + \frac{\partial^2\phi}{\partial r^2} + \frac{2}{r}\frac{\partial\phi}{\partial r} + \frac{1}{r^2}\Delta_{\theta\varphi}\phi = \sigma;$$

with

$$\Delta_{\theta\varphi} \equiv \frac{\partial^2}{\partial\theta^2} + \frac{1}{\tan\theta}\frac{\partial}{\partial\theta} + \frac{1}{\sin^2\theta}\frac{\partial^2}{\partial\varphi^2}$$

In 1D, it admits two characteristics : $\pm c$: $f(ct-x)$ and $f(ct+x)$.
To be well-posed, the initial-boundary value problem needs :

- $\phi(t=0)$ and $\partial\phi/\partial t(t=0)$,
- a boundary condition at every domain boundary (Dirichlet, von Neumann, mixed).

An explicit scheme for the wave equation

Using a second-order scheme to evaluate the second time derivative

$$\left.\frac{\partial^2\phi}{\partial t^2}\right|_{t=t^J} = \frac{\phi^{J+1} - 2\phi^J + \phi^{J-1}}{\delta t^2} + O\left(\delta t^4\right),$$

one recovers the forward Euler scheme

$$\phi^{J+1} = 2\phi^J - \phi^{J-1} + \delta t^2\left(\Delta\phi^J + \sigma\right) + O(\delta t^4).$$

Solution of the initial-boundary value problem inside a sphere or $r \leq R$:

- initial profiles at $t = t^0$ and $t = t^1$,
- $\forall t > t^1$, a value for $\phi(r = R)$.

With spectral methods using Chebyshev polynomials in r, time-step limitation is coming from the second radial derivative :

$$\delta t^2 \leq K/N^4.$$

Complete 3D problem $\Rightarrow$regularity conditions at the origin too, for $\ell > 1$.

Implicit scheme

With the same formula for the second time derivative and the Crank-Nicholson scheme :

$$\left[1 - \frac{\delta t^2}{2}\Delta\right] \phi^{J+1} = 2\phi^J - \phi^{J-1} + \delta t^2 \left(\frac{1}{2}\Delta\phi^{J-1} + \sigma^J\right).$$

One must invert the operator $1 - 1/2\delta t^2\Delta$; one way is :

- consider the spectral representation of ϕ in terms of spherical harmonics ($\Delta_{\theta\varphi} Y_\ell^m = -\ell(\ell+1)Y_\ell^m$) ;
- solve the ordinary differential equation in r as a simple linear system, using *e.g.* the tau method.

⇒one can add boundary and regularity conditions depending on the multipolar momentum ℓ.
⇒beware of the condition number of the operator matrix !
⇒sometimes regularity is better imposed (stable) using a Galerkin base.

Domain boundaries

Contrary to the Laplace operator Δ, the d'Alembert one $\Box$ is not invariant under inversion / sphere.

- one cannot *a priori* use a change of variable $u = 1/r$!
- the distance between two neighboring grid points becomes larger than the wavelength...

⇒domain of integration bounded (*e.g.* within a sphere of radius R).

Two types of BCs :

- reflecting BC : $\phi(r = R) = 0$,
- absorbing BC...

An absorbing BC can be seen in 1D : at $x = 1$ one imposes no incoming characteristic $\Rightarrow$only $f(ct - x)$ mode.
In spherical 3D geometry : asymptotically, the solution must match

$$\phi \sim_{r\to\infty} \frac{1}{r} f(ct - r),$$

equivalently,

$$\lim_{r\to\infty} \frac{\partial(r\phi)}{\partial t} + c\frac{\partial(r\phi)}{\partial t} = 0.$$

At finite distance R :

$$\left(\frac{1}{c}\frac{\partial\phi}{\partial t} + \frac{\partial\phi}{\partial r} + \frac{\phi}{r}\right)\Big|_{r=R} = 0;$$

which is exact in spherical symmetry.

General form of the solution

The homogeneous wave equation $\Box\phi = 0$ admits as asymptotic development of its solution

$$\phi(t, r, \theta, \varphi) = \sum_{k=1}^{\infty} \frac{f_k(t - r, \theta, \varphi)}{r^k}.$$

One can show that the contribution from a mode ℓ exists only for $k \leq \ell + 1$. Moreover, the operators :

$$B_1 f = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial r} + \frac{f}{r}, \qquad B_{n+1} f = \left(\frac{\partial}{\partial t} + \frac{\partial}{\partial r} + \frac{2n+1}{r}\right) B_n f$$

are such that the condition $B_n\phi = 0$ matches the first n terms ($B_n\phi = O\left(1/r^{2n+1}\right)$). It follows that

$$B_n\phi = 0$$

- is a n^{th}-order BC,
- is exact for all modes $\ell \leq n - 1$,
- is asymptotically exact with an error decreasing like $1/R^{n+1}$,
- is the generalization of the Sommerfeld BC at finite distance ($n = 1$).

Absorbing BC for $l \leq 2$

The condition $B_3\phi = 0$ at $r = R$ writes

$$\forall (t,\theta,\varphi), \quad B_1\phi|_{r=R} = \left(\frac{\partial}{\partial t} + \frac{\partial}{\partial r} + \frac{1}{r}\right)\phi(t,r,\theta,\varphi)\Big|_{r=R} = \xi(t,\theta,\varphi),$$

with $\xi(t,\theta,\varphi)$ verifying a wave-like equation on the sphere $r = R$

$$\frac{\partial^2\xi}{\partial t^2} - \frac{3}{4R^2}\Delta_{\theta\varphi}\xi + \frac{3}{R}\frac{\partial\xi}{\partial t} + \frac{3\xi}{2R^2} = \frac{1}{2R^2}\Delta_{\theta\varphi}\left(\frac{\phi}{R} - \frac{\partial\phi}{\partial r}\Big|_{r=R}\right).$$

- easy to solve if ξ is decomposed on the spectral base of spherical harmonics !
- looks like a perturbation of the Sommerfeld BC...
- exact for $\ell \leq 2$ and the error decreases as $1/R^4$ for other modes.

I. WAVE EQUATION

The aim is to solve the three-dimensional homogeneous wave equation $\Box\phi = 0$ in a sphere of radius R, using spherical coordinates:

$$\frac{1}{c^2}\frac{\partial^2\phi}{\partial t^2} - \frac{\partial^2\phi}{\partial r^2} - \frac{2}{r}\frac{\partial\phi}{\partial r} - \frac{\Delta_{\theta\varphi}\phi}{r^2} = 0. \tag{1}$$

Here, $\Delta_{\theta\varphi}$ is the angular part of the Laplacian. In what follows $c = 1$ is assumed. There shall be possibly three types of boundary conditions (BC) to be implemented:

1. Homogeneous BC : $\phi(r = R) = 0$, which models the reflection on the boundary.
2. Sommerfeld BC : $\partial(r\phi)/\partial t + \partial(r\phi)/\partial r|_{r=R} = 0$, which models a transparent boundary (at least for $\ell = 0$ wave modes).
3. Enhanced outgoing BC : $\partial(r\phi)/\partial t + \partial(r\phi)/\partial r|_{r=R} = \xi(\theta, \varphi)$, which is analogous to the Sommerfeld BC, but is also transparent to $\ell = 1, 2$ wave modes. The function $\xi(\theta, \varphi)$ verifies a wave-like equation on the boundary (see Sec. VI).

II. EXPLICIT SOLVER

The constant time-step is noted dt and $\phi^J = \phi(J \times dt)$, where the spatial coordinates are skipped. The simple forward Euler scheme writes:

$$\phi^{J+1} = 2\phi^J - \phi^{J-1} + dt^2\Delta\phi^J + O(dt^4). \tag{2}$$

This scheme can be safely used for small time-steps and spherical symmetry ($\ell = 0$ only).

Second-order time discretisation of the Sommerfeld BC writes:

$$\left(\frac{3}{2dt} + \frac{1}{R}\right)\phi^{J+1}(R) + \left.\frac{\partial\phi^{J+1}}{\partial r}\right|_{r=R} = \frac{4\phi^J(R) - \phi^{J-1}(R)}{2dt} + O(dt^2). \tag{3}$$

III. SUGGESTED STEPS

- Setup a spherically-symmetric one-domain grid (`Mg3d`, but only nucleus), with a mapping and associated r coordinate.
- Define an initial profile for ϕ^0 and ϕ^1 (*e.g.* the same Gaussian one for both), which should be of type `Scalar`.
- Make a time loop for 2-3 grid-crossing times with a graphical output (with the function `des_meridian`, see LORENE documentation).
- Doing so, the problem is ill-posed and therefore unstable. Add the BC requirement (homogeneous or Sommerfeld BC) by modifying at each time-step the value in physical space of the point situated at $r = R$, with the method `Scalar::set_outer_boundary`. Note that the initial profile must satisfy the BC!
- Make runs with varying the time-step to see the Courant limitation.

IV. IMPLICIT SOLVER

The 3D extension of the previous approach is very uneasy, it is therefore recommended to used implicit schemes, namely the Crank-Nicholson one:

$$\left[1 - \frac{dt^2}{2}\Delta\right]\phi^{J+1} = 2\phi^J - \phi^{J-1}\frac{dt^2}{2}\Delta\phi^{J-1} \tag{4}$$

The angular part of the Laplacian $\Delta_{\theta\varphi}$ admits spherical harmonics as eigen-functions:

$$\Delta_{\theta\varphi}Y_\ell^m = -\ell(\ell+1)Y_\ell^m \tag{5}$$

so that when developing ϕ onto spherical harmonics, the operator in (4) becomes

$$1 - \frac{dt^2}{2}\left(\frac{\partial^2}{\partial r^2} + \frac{2}{r}\frac{\partial}{\partial r} - \frac{\ell(\ell+1)}{r^2}\right) \tag{6}$$

for each harmonic.

V. SUGGESTED STEPS

- Take a symmetric grid (in θ and φ), with x and y coordinate fields, to define an $\ell \leq 2$ initial profile (*e.g.* $xy\times$ a Gaussian).
- At every time-step after transforming to Y_ℓ^m, make a loop on ℓ, m (use `Base_val::give_quant_numbers` to get ℓ and m) and build the matrix associated with the operator (6), acting on coefficient space, using elementary operators `Diff`. Be careful to take into account the mapping!
- Within the same loop on ℓ, m, fill a `Tbl` with the coefficients of the right-hand side of (4).
- Add the BC and a regularity condition (when necessary) using the tau method.
- Invert the system to get ϕ^{J+1}, go back to Fourier coefficients and, eventually, compute the energy stored in the grid:

$$E = \int \left(\frac{\partial \phi}{\partial t}\right)^2 + (\nabla \phi)^2 \tag{7}$$

  using the method `Scalar::integrale`.

VI. ENHANCED BOUNDARY CONDITIONS

These are a modification of the Sommerfeld BC (Sec. I), with $\xi(\theta, \varphi)$ verifying:

$$\frac{\partial^2 \xi}{\partial t^2} - \frac{3}{4R^2}\Delta_{\theta\varphi}\xi + \frac{3}{R}\frac{\partial \xi}{\partial t} + \frac{3\xi}{2R^2} = \frac{1}{2R^2}\Delta_{\theta\varphi}\left(\frac{\phi}{R} - \left.\frac{\partial \phi}{\partial r}\right|_{r=R}\right); \tag{8}$$

When developing ξ and ϕ onto Y_ℓ^m and using again Crank-Nicholson time scheme:

$$\begin{aligned}\frac{\xi_{\ell m}^{J+1} - 2\xi_{\ell m}^{J} + \xi_{\ell m}^{J-1}}{dt^2} &+ \frac{3}{8}\frac{\ell(\ell+1)}{R^2}\left(\xi_{\ell m}^{J+1} + \xi_{\ell m}^{J-1}\right) + \frac{3}{R}\frac{\xi_{\ell m}^{J+1} - \xi_{\ell m}^{J-1}}{2dt} \\ &+ \frac{3}{4R^2}\left(\xi_{\ell m}^{J+1} + \xi_{\ell m}^{J-1}\right) = -\frac{\ell(\ell+1)}{2R^2}\left(\frac{\phi_{\ell m}^{J}(R)}{R} - \left.\frac{\partial \phi_{\ell m}^{J}}{\partial r}\right|_{r=R}\right),\end{aligned}$$

one gets a simple numeric linear equation in terms of $\xi_{\ell m}^{J+1}$, which is to be solved at every time-step.

Implement this BC and test it against the Sommerfeld one either by doubling the grid, or by looking at the energy left inside the grid.

C++ digest

December 9, 2005

Contents

1 Introduction

This document is intended to give a few references on the way the C++ language works, for the people attending the school on spectral methods in Meudon in November 2005 and having no knowledge of this programming language. It is very brief (incomplete!) and deals only with most basic syntax and notions (classes, derived classes, virtual methods), with a few examples given in the last section. Interested persons can, of course, consult more complete manuals, like *The C++ Programming Language* by the designer of C++ Bjarne Stroustrup, or refer to one of the many available Web sites, like *e.g* `http://www.cplusplus.com`, which is quite useful when dealing with inputs/outputs (iostream library).

One should not focus too much on the fact that C++ is called an "object-oriented language". It is a programming language with *function* calls and use of *variables*, which can be of different *types*. The notion of class simply gives to the programmer the possibility of defining and introducing his own types; as well as the associated functions to act and interact with other existing types.

As general remarks, it is necessary to declare all classes, functions and variables before they are used or implemented. Except for variables, these declarations are usually put to *header* (or *declaration*) files, which are then included into *source* files that uses or implements them. This implementation is often called *definition* of the function or class (see also Sec. 4). Then, a distinction is made between static and dynamic properties in the program: a *static* feature can be determined or resolved when then program is compiled; whereas a *dynamic* one is completely defined only when the program is executed (for example, it depends on some quantity given by the user at run time).

Finally, all instruction lines must end with a semicolon "`;`" ...

2 Basic syntax

For those who already know about C programming language, it is in principle possible to use any C instruction in C++ too. Although this might be very helpful for people having good skills for inputs / outputs in C, it is however recommended to switch to C++ syntax, as far as memory allocation is concerned, and to forget about `malloc` or `free`...

2.1 Types and variables

A *variable* is the basic brick of the program: it contains information to be processed like *e.g.* numbers. Indeed, some of the simplest *types* of variable, that are pre-defined in C++ are `int`, `float`, `double`, `char`, `bool`, ...:

- `int n ; n = 5 ;` integer number;
- `float x=1.5e10F ;` real floating-point number in single precision; stored on 4 bytes, it describes 8 digits (here $x = 1.5 \times 10^{10}$);
- `double y ; y=2.3e-9 ;` real floating-point number in double precision; stored on 8 bytes, it describes 16 digits (here $y = 2.3 \times 10^{-9}$);
- `char l='w';` a single character (`'\n'`is a newline, `'\t'`a tabulation, ...);
- `bool f=true;` boolean variable that can only be `true` or `false`.

The word "`const`" in front of a type name means that the variable is *constant* and thus its value cannot be modified by the program. Constant variables are initialised when they are declared:

```
const int n = 8 ; // now, writing 'n=2;' is forbidden!
```

Variables can be declared at any point in the code, provided that, of course, they are declared before they are used. The declaration is valid only within the local *block*, *i.e.* within the region limited by braces ("`{ }`"). In the example of Sec. 4.1, the variable `square` is defined only until the first "}", two lines after. It is the local *declaration scope* of variables in C++.

2.2 Pointers and references

A *pointer* on a variable is the address where this variable is stored in the system memory. Pointers can then also be used as variables in the program.

```
int n ;
n = 2 ;
int *p ;
p = &n ;
int x = *p ;
```

In this example, `p` is declared on third line as a variable of type "pointer on an integer" (the type is `int *`); the line after, it is initialised to be the address of the variable `n` (the ampersand meaning "the address of"). Finally, if one wants to use the value stored at the address defined by `p`, a star (`*`) must be put in front of the pointer: on the last line, `x` is initialised to 2. One can therefore see two ways of manipulating variables: through their *values* (like `n` or `x` in this example), or through their addresses (like `p`).

In C++, there is a third way to do so: the *references*. A reference to a variable can be seen as an equivalent to this variable: if one of the couple (variable / reference) is modified, the other is changed too.

```
int n ;
n = 3 ;
int &r = n ;
r++ ; //equivalent to r = r + 1 ;
int x = r - n ;
```

Here, `r` is declared as a reference to an integer (type `int &`) and must be immediately initialised, like `const` variables. In the fourth line, `r` is incremented, and so is `n`, since they are equivalent: every modification to `r` also affects `n`! The results is that, on the last line, `x` is initialised to `4-4=0`.

Constant pointers have a rather different meaning from constant values. `const double *x;` means that the variable pointed by `x` is constant, not `x`! This means that one can write:

```
int n=3;
int p=4;
const int *pn = &n ;
pn = &p ; // OK
*pn = 9 ; //forbidden!
```

One can change the address `pn` (in this case, from the address of `n` to the address of `p`), but one can access the variable pointed by this address in readonly mode. In order to have a constant address, the following syntax must be used:

```
int *const pp = &p ;
```

in that case, an instruction of the type `pp = &n;` is forbidden. Thus, writing `const int *const pq = &n ;` means that both the address `pq` and the variable stored at this address are constant. As for references, only the syntax `const double &x;` has a meaning: `x` can be accessed in readonly mode.

2.3 Functions

In C++, a *function* is a name and a list of arguments; eventually it returns something. All parts of the code (main program, subprograms) are functions and, apart from the main one, they must all be declared before there is a call to them. This declaration is only the statement of (from left to right): the return type (if the functions returns nothing, then it is `void`), the name of the function and, in between parenthesis, a list of the types of its arguments. If there is no argument, then empty parenthesis `()` should be used. An example can be seen in Sec. 4.1, with the function `my_function`, returning a `double` and having as arguments a `double` and an `int`. It is declared before the call in the function `main` (the main program, which always returns an `int`, and which is compulsory to have a code running); it is then defined after this main program. Note that the function `main` does not need to be declared.

Some arguments can be set to default values in the declaration of a function. Consider, for example the declaration of a function that displays an integer in a given base:

```
void display_integer(int num, int base=10) ;
```

so that it is called `display_integer(128, 16)`, to display `128` in hexadecimal base. But with the declaration done above, one can call it also `display_integer(20)` to display `20` in decimal base. The second argument has a *default value*, that need not be specified at the function call. Such arguments (there can be several of them) must always be located at the end of the argument list and, if an argument with a default value has its value specified, all those which are before him must also have their values specified. Please note that default values appear only in the declaration and never in the definition of a function.

As stated at the beginning of this section, a function is specified not only by its name, but also by the list of the types of its arguments. Therefore, it is possible to *overload* a function with another having the same name, but different arguments:

```
int sum(int a, int b) ;
double sum(double x, double y) ;
```

When the function `sum` is called, the choice is made (when compiling) looking at the arguments' types. Note that the return type is not discriminatory and a function `double sum(int, int)` cannot be declared together with the first one of the here-above example.

To end with functions, a few details about arguments and return values are given. A variable given as an argument to a function is not "passed" in the strict sense: the function makes a *copy* of the variable and works on that copy. This means that a function cannot modify a variable, passed as argument! This can be seen on the following example:

```
void swap(int a, int b) { int c=a;
a=b; b=c ; }
int main() {
int n = 2 ; int p = 3 ;
swap(n,p) ; }
```

After the call to `swap`, `n`=2 and `p`=3 still! The way of doing here is to pass the variables by their addresses or references on them.

```
void swap(int *a, int *b) { int c= *a;
*a=*b; *b=c; }
int main() {
int n = 2 ; int p = 3 ;
swap(&n,&p) ; }
```

or

```
void swap(int &a, int &b) { int c=a;
a=b; b=c ; }
int main() {
int n = 2 ;  int p = 3 ;
swap(n,p) ; }
```

... and everything works fine, since a copy of the address (or reference) still points on (refers to) the same variable. Therefore, if a variable is an output argument of a function, it should not be passed by its value.

2.4 Tests, loops and operators

A simple *test* is written as follows:

```
if (a > 5) {
b = 5 ; //etc ...
}
else { // if needed
b = 3 ;
}
```

The expression following the word `if` must be surrounded by parenthesis and of boolean type: `(n == 2)` equality test, which is different from assignment and has two `=`'s, `(p != 0)` different from, `(x >= 2.e3)`, ... or a combination of such, using `&&`(logical "and") or `||`(logical "or"). If the condition is `true`, then the block following it is executed. Eventually, one can put an instruction (followed by a block) `else{...}`. Several other tests are available:

- `do {...} while` (some test)
- `while` (some test) `{...}`
- some test `?` action1 `:` action2 `;`

In this last case (conditional operator), if "some test" is `true` then "action1" is executed, otherwise, it is "action2".

The operator *switch* is used to choose between different instructions depending on the value of an integer-like variable:

```
switch (n) {
case 0 :
 p = 2 ;
 break ;
case 1 :
 p = 3 ;
 break ;
default:
 p = 0 ;
 l = 3 ;
 break ;
}
```

Finally, a test that is often useful is given by the *assert* command:

```
#include<assert.h>
void my_log(double a) {
assert(a > 0.) ;
... }
```

After the inclusion of its declaration in the file `assert.h`, the use is `assert(` boolean expression `)`. If the boolean expression is true, then the code continues to the next line; otherwise the code is stopped by an `abort()` command. The advantage is that, when compiling the code with the `-DNDEBUG` option, the assert tests are not performed and thus, for an optimised version, it is costless.

The syntax for a *loop* is quite simple and needs three instructions :

1. the initialisation of the loop variable
2. the test (done before loop instructions)
3. the increment of the loop variable (each time the loop is ended)

```
for (int i=0; i<20; i++) {
... //loop instructions
}
```

Here, `i` runs from 0 to 19; note that it is a variable local to the loop *i.e.* it is no longer valid after the loop.

It is also worth mentioning some of the *arithmetic operators*, the last ones in this table can shorten some expressions:

operator	`+, -, *, /`	`%`	`+=` (`a += 5;`)	`-=, *=, /=, %=`	`++` (`a++;`)
meaning	usual arithmetic	module[1]	`a = a + 5;`	similar to `+=`	`a = a + 1;`

2.5 Inputs / Outputs

This section deals with formatted *input*/*output* manipulations. These are done through input- or output-streams and the *iostream* library. The "standard output" (the shell console on which one is typing commands) is called *cout*, and one can send it data thanks to the injection operator "`<<`":

```
int a = 9 ;
cout << a ;
```

will display '9'. Strings can be displayed directly: `cout<<"Hello world!"<<endl;` where `endl` stands for a new line (and empties the buffer). More about strings is given in Sec. 2.6. All standard types (see Sec. 2.1) can be outputted in this way, without specification of the format to be used. In order to change the format (precision, fixed / scientific, *etc* ...), *manipulators* are employed (see `http://www.fredosaurus.com/notes-cpp/io/omanipulators.html`). Standard input (from the keyboard, in the console) is accessed through *cin*:

[1] module being the operation that gives the remainder of a division of two integer values

```
cout << "Enter a number:" << endl ;
int n ;
cin >> n ;
```

Any function using `cin`, `cout` or `>>`-like operators must have declared the iostream library and must use the standard namespace (see Sec. 2.7) by adding the following two lines before its definition:

```
#include<iostream>
using namespace std ;
```

Accessing to files is done in a very similar way:

```
double x = 1.72e3 ;
ofstream my_file("output.dat") ;
my_file << "The value of x is: " << x << endl ;
```

this opens a file called "output.dat" (created, if it does not exist, erased if it does) and writes things into it. `my_file` is an object of type *ofstream* (output file stream), linked with the file opened in write-mode. Similarly, to read data from a formatted (existing) file, one should do as for reading from the standard input:

```
ifstream a_file("input.dat") ;
int a ;
a_file >> a ;
```

In this case, `a_file` is of type *ifstream* (input file stream). Before using any of these types, one should declare, in addition to the iostream library and the standard namespace, the fstream one with a "`#include<fstream>`".

2.6 Memory allocation

There are two ways of defining an *array* in C++: static and dynamic allocation. The static way is *e.g.* `double tab[37] ;` (note that the indices of `tab` range from 0 to 36), or with constant integer variable for the dimension `const int nsize = 100; int tbl[nsize];`. Here the size of each array is known at compilation time. On the contrary, when this size cannot be known, one must use *dynamic memory allocation*:

```
int n ;
cout << "Enter size" << endl ;
cin >> n ;
double *tab = new double[n] ;
```

In this case, `tab` is an array, which elements can be accessed as before: from `tab[0]` to `tab[n-1]`. The syntax used here shows that an array can be seen as a pointer on its first element, so there is an implicit compatibility between arrays and pointers. The allocation of the memory is done at runtime thanks to the instruction *new*, but this memory must then be given back to the system, using the instruction *delete*, when the array is no longer used: `delete [] tab;`

The simplest implementation of *strings* is done through arrays of `char`s, with the use of double quotes, contrary to single `char`s:

```
char* var = "Hello!";
```

another example is given is Sec. 4.1. Note that such strings end with the character `'\0'`, so here `var` has seven elements.

A full program implementing many features described here is shown is Sec. 4.1.

2.7 Static variables and namespaces

A *static variable* keeps its value from one function call to the next:

```
void f() {
static int n_call = 0 ;
if (n_call == 0) { ... } //first call operations
n_call++ ; ...
```

In this example, `n_call` is initialised to 0 when the function is called for the first time, but it then keeps its value (it is no longer initialised!) at next function calls. So, in this case, the value of `n_call` is the actual number of calls to `f()`.

However, this kind of syntax can be replaced in C++ with the use of a *namespace*. This is a declaration region that can be used in several functions, without interfering with local variables:

```
namespace my_name {
int i, a ;
void g() ;
}
```

Here is declared a namespace called `my_name` with `i, a` and the function `g` as members. After including the file containing this namespace, one can use some of its variables `my_name::i = 0;`, or the whole namespace:

```
using namespace my_name ;
g() ;
```

In such case there is no need to specify `my_name::`, the call to `g()` means `my_name::g()`. In LORENE, namespaces are used to carry unit definitions (numerical constants). Finally, in order to replace static variables, one should use an *anonymous namespace*:

```
namespace {
int n_call = 0 ;
}
void f() {
if (n_call ... }
```

in particular, there is no instruction `using`.

3 Classes

A *class* is a collection of data and functions. It generalises the notion of type by giving the possibility to the programmer to define new types of his own. In particular, one can overload (see Sec. 2.3) standard operators (*e.g.* arithmetic operators, or output) with these new types, as it is shown in complete examples of Sec. 4.2 and 4.3. Once a class is declared and defined, one can declare variables of that new type, use pointers on it or references to it, as if it were a standard type (double, int, ...). Classes has already been used in this document: output files were declared as `ofstream` in Sec. 2.5, which is a C++ class...

3.1 Members

To declare a class, one must specify its *members*: data (variables of other types, including eventually other classes) and functions, sometimes called *methods*. The syntax is:

```
class My_class {
int n ;
double x ;
double f(int) ;
} ;
```

This declares a class called `My_class`, with two data members, called `x` and `n`, and one method `f(int)`. Actually, the full name of the method is (to be used when defining it outside the class declaration):

```
double My_class::f(int p) {
x = 2.3*(p+n) ;
double res = 3./x ;
...}
```

Methods of a class can use the data without need to re-declare them: here `n` or `x` are known to be members of `My_class`.

In a function using this class, these members are used the following way (one must include the declaration of the class before using it)[1]:

```
My_class w ;
int q = w.n + 2 ;
cout << w.f(q) ;
```

The variable `w` is an object of type `My_class`, and has its own members that can be accessed through the operator "`.`". When considering pointers, the operator is "`->`":

```
My_class *v ;
double y = v->x + 0.2 ;
cout << v->f(3) ;
```

Usual arithmetic operators can be overloaded to work with this new class:
e.g. `My_class operator+(My_class, My_class);` for the declaration, and, in some other function:

```
My_class z1 ;
My_class z2 ;
My_class t = z1 + z2 ;
```

3.2 Restriction of access and `const` issues

A central notion when manipulating classes is that some of its members are not accessible (*i.e.* usable) by "normal" functions, (functions which are not member of the same class. This is called *restriction of access* and is achieved through the keywords `public`, `private` and `protected`:

[1] the first line assumes that there is a constructor (see Sec. 3.3) without parameters

```
class My_class {
private:
int n ;
double x ;
public:
double f(int) ;
private:
void g() ;
} ;
```

In this case, data and the method `My_class::g()` are private, meaning that only functions member of `My_class` can use them, therefore the last two lines of

```
My_class v ;
cout << v.n ;
v.g() ;
```

are forbidden in all other functions. On the contrary, a call to `v.f(2)` is allowed. By default, all members of a class are private, so it is highly recommended to use the keywords when declaring a class. Such a keyword is valid until the use of another one; and `protected` has the same effect as `private` in a class, the difference appearing only for derived classes (Sec. 3.4).

Exceptions can be made declaring some functions or other classes to be *friend*, within the declaration of the class:

```
class My_class {
private:
int n ; ...
friend double ext_f() ;
} ;
```

Then, inside the definition of the non-class member `double ext_f()`, one can access private and protected members of `My_class`. A class can be declared friend the same way: adding `friend class Other_class ;` in the declaration of `My_class` and, inside all methods of `Other_class`, it is then possible to access to private/protected members of `My_class`.

The notion of *constant* object (see Sec. 2.1) means for a class that all data of this object are constant. Exceptions are possible with the keyword `mutable`: a *mutable* data member is a member that can be modified, although the object it belongs to is seen as constant. The idea can be that mutable members are somehow "secondary" members, that can be deduced from other "primary" data. So, as long as these primary data are not modified, the object is supposed constant. The syntax is, within a class declaration:

```
class My_class {
public:
int n; ...
mutable double sec ;
} ;
```

So, in a function using `My_class`, one has then:

```
const My_class w = ... ;
w.n = 2 ;                // forbidden!
w.sec = 0.7 ;            // OK
```

A member function that keeps the object it is called on constant is said constant too, the keyword `const` can be added at the end of its declaration and definition:

```
class My_class {
...
double f(int) const ;
void g() ;
... };
```

Here, `double My_class::f(int)` is constant, whereas `void My_class::g()` is not. As before, in a function using this class:

```
const My_class w = ... ;
cout << w.f(2) ;           //OK
w.g() ;                    //forbidden!
```

3.3 Constructors, destructor and assignment operator

When designing a new class, special care must be devoted to four particular member functions:

- a *standard constructor* – this function is called to create an object of this class. It has the same name as the class, and no return type (*e.g.* `My_class::My_class()`. It can take arguments or not and its task can be (not compulsory at all) to initialise data members or to allocate memory.
- a *copy constructor* – creates an object from an existing one. It is also a constructor as the standard one but it must take exactly one argument of the type `const My_class &`, meaning that it needs a reference on an object of that class that will not be modified to build the new object (readonly!).
- a *destructor* – destroys the object when it is no longer valid, *i.e.* at the end of its declaration scope. Its task is mainly to check if there is some dynamically allocated memory to be given back to the system. It has the same name as the class, but with a tilde (~) in front; it takes no argument and has no return type.
- an *assignment operator* – used to assign an existing object to another one, with an instruction like `a=b`. Its name is `My_class::operator=` and, as the copy constructor, it takes one argument of the type `const My_class &`.

These four methods are compulsory to have a usable class. Note that there can be several constructors as long as there is a copy constructor and a standard one. A "fifth" function is very useful: an overload of the `<<` operator to display objects of the class. This function is necessarily an external one (*i.e.* it is not a member of the class) and can also be used to output to files instead of `cout`. A full example implementing all these functions, together with a main function is shown in Sec. 4.2, with the class `rational`, representing rational numbers. In particular, in the definition of this class (Sec. 4.2.2), both constructors use the *initialisation list* that directly initialises class data from the constructor's arguments

```
rational::rational(int a, int b) : num(a), denom(b) { ...}
```

Here, data `num` and `denom` are directly copied from the variables `a` and `b`; *i.e.* it is equivalent to writing

```
 rational::rational(int a, int b) {
num = a ;
denom = b ; ... }
```

Still, the initialisation list is more readable and will be used for derived classes (see next section).

Another example is given by the class `My_array` in Sec. 4.3, with dynamic memory allocation (*i.e.* a non-trivial destructor). Only the main structure is given, but the class can be compiled.

3.4 Derived classes

An existing class can be completed into a new class, which then has more members. This is the *inheritance* mechanism that allows the (new) *derived class* to get the properties of the (existing) *base class*, and to add new ones. The declaration of the new class is done as follows (once the class `My_class` has been declared):

```
class New_class : public My_class {
int p ;
int h() ; }
```

In this example, the class `New_class` is derived from `My_class`, to which it adds two new members.

Actually, the derived class does not inherit all members of the base class. First, the private members are not accessible to the derived class methods (whereas protected ones are! This is the difference between private and protected access); then, none of the assignment operators (which name are `operator=`), constructors or destructor are inherited. Therefore, one must re-declare and re-define these members, but with the help of their equivalents in the base class. In particular, for the constructors of the derived class, one must first call the constructor for the base class, through the initialisation list. Then, the new members of the derived class are initialised in this list and, finally, other actions are performed in the body of the constructor. The destructor for the new class works in the opposite way: first the new members must have their memory given back to the system (if any) and, at the end, there is automatically a call to the destructor of the base class. A simple example is given in Sec. 4.3.4, with the new class `Square_matrix` being a derived class from `My_array`, but with no new data member.

A very important point is that there is an *implicit compatibility* between the derived class and the base class. This is valid only for pointers and references to the derived class, which can be used instead of pointers or references to the base class.

```
My_class *w ;
New_class *q ;
w = q ;
```

is allowed, whereas `q = w;` is forbidden. After the third line above, the static type of `w` is `My_class *` (obtained from the declaration), whereas the dynamic type is `New_class *`, since it is pointing on an object of this derived class. This dynamic type cannot, in general, be determined at compilation time (imagine there is a test depending on some reading from `cin`, to decide whether `w = q` is invoked or not). If one wants to know, in this example, the type of `w`, a possibility is to use the instruction *dynamic_cast*:

```
New_class *z = dynamic_cast<New_class *>(w) ;
```

In that case, if the dynamic type of `w` is `New_class *`, then `z != 0x0` (`z` is not the null pointer). Actually, this is the case if `w` is compatible with the type; *i.e.* `z` is not null also if `w` is a pointer on a derived class of `New_class`.

The equivalent for references can be seen in the assignment operator of `Square_matrix` (Sec. 4.3.4), where there is a call to the assignment operator of the base class `My_array`, but with a reference to an object of the derived class as argument, instead of a reference to an object of the base class (see the declaration of `My_array::operator=`).

3.5 Virtual methods

In the examples cited above, a problem can arise if the derived class re-declares a method of the base class:

```
class My_class {
double f(int) ; ...} ;
class New_class : public My_class {
double f(int) ; //different from that of My_class
...} ;
My_class *w ;
New_class *q ;
int v ; cin >> v ;
if (v==0) {
w = q ; }
cout << w->f(2) ;
```

Which method `f(int)` is called ? This is impossible to determine at compilation time, but is not an academical problem since, in each derived class, this is exactly the case for destructors. In the above example, it might happen that not all the memory allocated to `w` is freed. Therefore, there is a mechanism in C++ called *polymorphism* that makes the link with the right function at execution time (dynamically). It is obtained by the use of the keyword *virtual*, for the declaration, in front of such "ambiguous" methods:

```
class My_class {
virtual double f(int) ; ...} ;
class New_class : public My_class {
virtual double f(int) ;
...} ;
```

Now, everything works fine and the call is done to the right function. The only requirement is that the list of arguments must be the same for all the virtual functions having the same name. Note that, every time inheritance is used, one must declare all destructors of base / derived classes as `virtual`. Another example is given in Sec. 4.3.3 with the method `display(ostream&)`: the standard display is achieved through the call to `operator<<` which, thanks to implicit compatibility, can also be called with (reference to) `Square_matrix` objects. This function then calls to the virtual method `display(ostream&)`, which gives different output, depending on the type of `tab_in`.

3.6 Abstract classes

With the possibility of deriving classes, it is sometimes interesting to have some classes that are not actually usable, but that can be used as templates for the design of other classes. These

classes therefore possess one or several methods that are too general to be defined (implemented): in LORENE, this is the case *e.g.* for a general equation of state. Such kind of functions are then declared as *pure virtual method*. The declaration is then ended with a "`=0 ;`" and no definition is given:

```
 class Eos {
virtual double p_from_rho(double) = 0 ; ... };
```

Still, a derived class, which is usually more specific, can implement that method, using polymorphism:

```
 class Eos_polytrope: public Eos {
virtual double p_from_rho(double)  ; ... };
```

In the example of `Eos`, one cannot declare an object of this type, since the class is incomplete, only a derived class which implements the pure virtual methods can be used. Nevertheless, one can declare a pointer or a reference to an `Eos`:

```
Eos eo ; //forbidden
Eos_polytrope ep ; //allowed, it implements p_from_rho
Eos *p_eos = &ep ; // OK, not instance +  implicit compatibility
Eos &r_eos = ep  ; // OK, not instance +  implicit compatibility
```

`Eos` is called an *abstract class*, for one cannot declare any *instance* (no direct objects, only pointers or references to) of this class. More generally, since an abstract class is a class which cannot be instantiated, these are classes that:

- have a pure virtual method;
- derive from a class with a pure virtual method that they do not define;
- have only private or protected constructors.

4 Examples

4.1 A first program

This program does not do any interesting job, it is just an illustration of the basic syntax in C++.

```
// C++ headers
#include <iostream> //<> are for system headers, "" for user-defined ones
#include <fstream>
// C headers
#include <math.h> // in principle, C headers contain a .h, whereas C++ do not
using namespace std ;  // to get input / output objects (cin, cout, ...)
double my_function(double , int) ; // local prototype (declaration only)

int main(){  // In every executable there must be a main function returning an integer
  const int nmax = 200 ;
  double stat_array[nmax] ; //static allocation of memory
  char dim[] = "size" ;
```

```
  cout << "Please enter a "<< dim << " for an array between 1 and 200" << endl ;
  int dyn_size ;
  cin>>dyn_size ;
  if ((dyn_size<1)||(dyn_size>200)) {
    cout << "the " << dim <<" must be between 1 and 200!" << endl ;
    cout << "try again: " ;
    while ((dyn_size<1)||(dyn_size>200)) cin>>dyn_size ;
  }
  double *dyn_array = new double[dyn_size] ; //dynamic memory allocation
  for (int i=0; i<nmax; i++) {
    int square = i*i ;
    stat_array[i] = square ;
  }
  double cube ;
  for (int i=0; i<dyn_size; i++) {
    cube = pow(double(i),3) ; //Conversion of an integer to a double
    dyn_array[i] = cube + my_function(stat_array[i], dyn_size) ;
  }
  cout << "The value of the variable dyn_array is: " << dyn_array << endl ;
  cout << "its first element is: "<< *dyn_array
       << " or, alternatively: " << dyn_array[0] << endl ;
  cout << "Saving dyn_array to the file exa1.dat..." << endl ;
  ofstream output_file("exa1.dat") ;
  for (int i=0; i<dyn_size; i++) {
      output_file << i << '\t' << dyn_array[i] << '\n' ; }
  output_file << endl ;
  delete[] dyn_array ; // It is necessary to release the allocated memory...
  return EXIT_SUCCESS ; // If the program came up to here, everything went fine
}
// definition of "my_fonction"
double my_function(double x, int n) {
  double resu = log(x+double(n)) ;
  return resu ;
}
```

4.2 A class of rational numbers

To compile it, just type (*e.g.* with the GNU C++ compiler):

```
g++ -o ratio ratio.C rational.C gcd.C
```

4.2.1 Declaration file `rational.h`

```
#ifndef __RATIONAL_H_        // to avoid multiple declarations
#define __RATIONAL_H_
#include <iostream>          // ostream class is used
using namespace std ;
class rational {             // beginning of the declaration of class rational
    // Data:
    // -----
```

```
 private:
    int num ;                    //  numerator
    int denom ;                  //  denominator

    //Required member functions
    //-------------------------
 public:
    rational(int a, int b = 1) ; // Standard constructor to create a/b
    rational(const rational& ) ; // Copy constructor

    ~rational() ; // Destructor

    void operator=(const rational&) ; //Assignment from another rational

    // Data access
    // -----------
    int get_num() const {return num ; };       //inline definition
    int get_denom() const {return denom ; } ;  //inline definition

    //Display: declaration of "friendness" only
    friend ostream& operator<<(ostream& , const rational& ) ;
}; // end of the declaration of class rational

//True declaration of the function, not member of the class
ostream& operator<<(ostream& , const rational& ) ;
// External arithmetic operators to calculate expressions such as 'p + q*r'
rational operator+(const rational&, const rational&) ; // rational + rational
rational operator-(const rational&, const rational&) ; // rational - rational
rational operator*(const rational&, const rational&) ; // rational * rational
rational operator/(const rational&, const rational&) ; // rational / rational
#endif
```

4.2.2 Definition file `rational.C`

```
// Include files
#include <assert.h>
#include "rational.h"

int gcd(int, int) ; //local prototype of an external function (greatest common divisor)

//--------------//
// Constructors //
//--------------//
// Standard
rational::rational(int a, int b):num(a), denom(b) {
  assert(b!=0) ;

  if (num == 0) denom = 1 ;
```

```
  else {
    int c = gcd(a, b) ;
    num /= c ;
    denom /= c ;
  }
}

// Copy
rational::rational(const rational & rat_in): num(rat_in.num), denom(rat_in.denom)
{
  assert(rat_in.denom != 0) ;
  assert(gcd(num, denom) == 1) ;

}
//------------//
// Destructor //
//------------//
rational::~rational() { }

//------------//
// Assignment //
//------------//
// From rational
void rational::operator=(const rational & rat_in) {
  assert(rat_in.denom != 0) ;
  num = rat_in.num ;
  denom = rat_in.denom ;
  assert(gcd(num, denom) == 1) ;
}

//---------//
// Display //
//---------//
// Operator <<
ostream& operator<<(ostream& o, const rational & rat_in) {

    if (rat_in.denom == 1) o << rat_in.num ; //as a friend it can access private data
  else
    o << rat_in.num << "/" << rat_in.denom ;
  return o ;
}

                        //-----------//
                        // Addition  //
                        //-----------//

// rational + rational, not friend, must use access functions
rational operator+(const rational& t1, const rational& t2) {
```

```
  rational resu(t1.get_num()*t2.get_denom() + t2.get_num()*t1.get_denom(),
t1.get_denom()*t2.get_denom()) ;
  return resu ;
}
```

4.2.3 GCD function

```
int gcd(int a, int b) {
  if (a<b) {
    int c = a ;
    a = b ;
    b = c ;
  }
  int reste = a%b ;
  while (reste != 0) {
    a = b ;
    b = reste ;
    reste = a%b ;
  }
  return b ;
}
```

4.2.4 Main program `ratio.C`

```
//Declarations of the class rational
#include "rational.h"

int main(){
    rational p(420,315) ;   // 420/315, simplified by the constructor
    rational q(5) ;         // 5/1
    cout<< p + q<< endl ;   // call to operator+ and operator<<
  return EXIT_SUCCESS ;
}
```

4.3 Classes `My_array` and `Square_matrix`

Although the classes `My_array` and `Square_matrix` can be compiled, they have incomplete features to be used on some real example. Only declaration and definition are given for better clarity.

4.3.1 Declaration file my_array.h

```
#ifndef __MY_ARRAY_H_
#define __MY_ARRAY_H_

#include<iostream>
#include<assert.h>
```

```
using namespace std ;

class My_array {
    // Data :
    // ------
 protected:
    int size1 ;        //size in first dimension ...
    int size2 ;
    int size3 ;
    double* tableau ;  // the actual array

    // Constructors - Destructor
    // -------------------------
 public:
  explicit My_array(int dim1, int dim2=1, int dim3=1) ; //standard constructor
  My_array(const My_array&) ; //copy constructor

  virtual ~My_array() ; //destructor

    // Assignments
    // -----------
  void operator=(const My_array&) ; //assignment from another My_array

    // Data access (inline)
    // --------------------
  int get_size1() const {return size1 ; };
  int get_size2() const {return size2 ; };
  int get_size3() const {return size3 ; };

  double operator()(int i, int j=0, int k=0) const { //read-only access (const)
      assert ((i>=0) && (i<size1)) ;    // tests: are we beyond array bounds?
      assert ((j>=0) && (j<size2)) ;
      assert ((k>=0) && (k<size3)) ;
    return tableau[(i*size2 + j)*size3 + k] ;
  };

  double& set(int i, int j=0, int k=0) {  //read-write access (thanks to the reference!)
    assert ((i>=0) && (i<size1)) ;
    assert ((j>=0) && (j<size2)) ;
    assert ((k>=0) && (k<size3)) ;
    return tableau[(i*size2 + j)*size3 + k] ;
  };

 protected:
  virtual void display(ostream& ) const ; //to use polymorphism

  // External function to be called for the display
  friend ostream& operator<<(ostream&, const My_array& ) ;
```

```
};

ostream& operator<<(ostream&, const My_array& ) ;
#endif
```

4.3.2 Definition file my_array.C

```
#include<fstream>  // to manipulate file streams ...
#include<iomanip>  // ...  and output format.
#include "my_array.h"

My_array::My_array(int dim1, int dim2, int dim3) : size1(dim1), size2(dim2), size3(dim3),
 tableau(0x0) {
  assert((dim1>0)&&(dim2>0)&&(dim3>0)) ;
  tableau = new double[dim1*dim2*dim3] ;
}

My_array::My_array(const My_array& tab_in) : size1(tab_in.size1), size2(tab_in.size2),
      size3(tab_in.size3), tableau(0x0) {

    assert((size1>0)&&(size2>0)&&(size3>0)) ;
    int t_tot = size1*size2*size3 ;
    tableau = new double[t_tot] ;
    assert(tab_in.tableau != 0x0) ;
    for (int i=0; i<t_tot; i++)
      tableau[i] = tab_in.tableau[i] ;
}

My_array::~My_array() {
  if (tableau != 0x0) delete [] tableau ;
}

void My_array::operator=(const My_array& tab_in) {
  assert(size1 == tab_in.size1) ;
  assert(size2 == tab_in.size2) ;
  assert(size3 == tab_in.size3) ;
  assert(tab_in.tableau != 0x0) ;
  assert(tableau != 0x0) ;

  int t_tot = size1*size2*size3 ;
  for (int i=0; i<t_tot; i++)
    tableau[i] = tab_in.tableau[i] ;
}

void My_array::display(ostream& ost) const {
  assert(tableau != 0x0) ;

  ost << "My_array: \n";
```

```
  ost << size1 << "x" << size2 << "x" << size3 << " elements" << endl ;
  ost << setprecision(5) ;

  for (int i=0; i<size1; i++) {
      ost << "i=" << i << '\n' ;
      for (int j=0; j<size2; j++) {
  for (int k=0; k<size3; k++) {
      ost << tableau[(i*size2+j)*size3 + k] << '\t' ;
  }
  ost << endl ;
      }
      ost << endl ;
  }
  ost << endl ;
  return ;
}

ostream& operator<<(ostream& ost, const My_array& tab_in ) {
  assert(tab_in.tableau != 0x0) ;
  tab_in.display(ost) ;
  return ost ;
}
```

4.3.3 Declaration file matrix.h

```
#ifndef __SQUARE_MATRIX_H_
#define __SQUARE_MATRIX_H_
#include "my_array.h"

class Square_matrix: public My_array { //inherits from My_array

    // Constructors - Destructor
    // --------------------------
 public:
    explicit Square_matrix(int ) ;  //standard constructor
    Square_matrix (const Square_matrix& ) ; //copy constructor

    virtual ~Square_matrix() ; //destructor (virtual, as needed)

    // Assignment
    // -----------
    void operator=(const Square_matrix&) ; //assignment from another Square_matrix

 protected:
    virtual void affiche(ostream& ) const ; //Display (virtual)
};
```

```
#endif
```

4.3.4 Definition file matrix.C

```
#include<iomanip>           //to have the manipulator setprecision()
#include "matrix.h"

// Default constructor
//--------------------
Square_matrix::Square_matrix(int dim1) : My_array(dim1, dim1) {
  assert(dim1>0) ;
}

// Copy constructor
//-----------------
Square_matrix::Square_matrix(const Square_matrix& tab_in) : My_array(tab_in) {}

// Destructor (does nothing, since there is an implicit  call to ~My_array() )
//-----------
Square_matrix::~Square_matrix() {}

// Assignment operator
//--------------------
void Square_matrix::operator=(const Square_matrix& tab_in) {
  My_array::operator=(tab_in) ;
}

// Display
//--------
void Square_matrix::affiche(ostream& ost) const {
  assert(tableau != 0x0) ;

  ost << "Square_matrix  " << size1 << "x" << size2 << endl ;
  ost << setprecision(5) ;
  for (int i=0; i<size1; i++) {
    for (int j=0; j<size2; j++) {
      ost << tableau[i*size2 + j] << '\t' ;
    }
    ost << endl ;
  }
  ost << endl ;
}
```

Index

Nom	Institut	Country
Ali Istiaq	Academy of science, Beijing	China
Ansorg Marcus	AEI	Germany
Arpad Lukacs	Inst. Part. Nuc. Phys., Budapest	Hungary
Bejger Michal	CAMK, Warsaw	Poland
Bene Gyula	Eötvös university	Hungary
Bonning Erin	LUTH	France
Brassart Matthieu	LUTH	France
Buffa Raffaele	Udine university	Italy
Chan Chi-Kwan	Arizona university	USA
Cordero Isabel	University of Valencia	Spain
Deshingkar Shrirang	Harish-Chandra Res. Inst.	India
Dragomirescu Ioana	University of Timisoara	Romania
Fodor Gyula	MTA RMKI Budapest	Hungary
Forgacs Peter	Université de Tours	France
Harada Tomohiro	Kyoto university	Japan
Jaramillo Jose Luis	LUTH	France
Jouve Laurène	SAP, CEA	France
Khadekar Goverdhan	Nagpur university	India
Klein Christian	Tuebingen university	Germany
Lin Lap-Ming	LUTH	France
Loeffler Franck	AEI, Gölm	Germany
Manca Gian Mario	Parma university	Italy
Motoyuki Saijo	Southampton	England
Peter Ralf	University of Tuebingen	Germany
Pétri Jérome	MPI, Heidelberg	Germany
Reuillon Sébastien	Université de Tours	France
Rocher Jérémie	LUTH	France
Rusznyak Adam	MTA RMKI Budapest	Hungary