

# Function approximation with global functions

MATMEK-4270

Prof. Mikael Mortensen, University of Oslo

# Important topics of today's lecture

- Approximation of smooth functions  $u(x)$  using **global** basis functions
- Variational methods
  - The least squares method
  - The Galerkin method
- The collocation method
- Runge's phenomenon
- Legendre polynomials
- Spectral convergence

# Function approximation with global functions

Consider a generic function

$$u(x), \quad x \in [a, b],$$

where  $u(x)$  can be anything, like  $u(x) = \exp(\sin(2\pi x))$ ,  $u(x) = x^x$ , etc., etc.

We want to find an approximation to  $u(x)$  using

$$u(x) \approx u_N(x) = \sum_{k=0}^N \hat{u}_k \psi_k(x)$$

where  $\psi_k(x)$  is a **global** basis function defined on the entire domain  $[a, b]$ .

# In the approximation

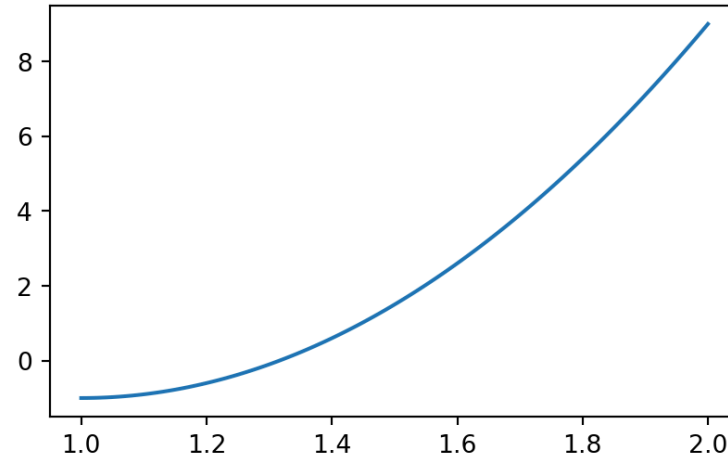
$$u_N(x) = \sum_{k=0}^N \hat{u}_k \psi_k(x) \quad (1)$$

- $\psi_k(x)$  is a basis function
- $\{\psi_k\}_{k=0}^N$  is a basis
- $\{\hat{u}_k\}_{k=0}^N$  are expansion coefficients (the unknowns)
- $V_N = \text{span}\{\psi_k\}_{k=0}^N$  is a function space (which is a vector space)

For example,  $V_N = \text{span}\{x^k\}_{k=0}^N$  is the space of all polynomials of order less than or equal to  $N$ . (More commonly referred to as  $\mathbb{P}_N$ )

If we say that  $u_N \in V_N$ , then we mean that  $u_N$  can be written as (1) and that  $u_N$  is a vector in the vector space (or function space)  $V_N$ .

**Example:**  $u(x) = 10(x - 1)^2 - 1$  for  $x \in [1, 2]$



Let  $V_N = \{1, x\}$  be the space of all straight lines.

What is the best approximation  $u_N \in V_N$  to the function  $u(x)$ ?

How do we decide what is best? All we know is that

$$u_N(x) = \hat{u}_0 + \hat{u}_1 x.$$

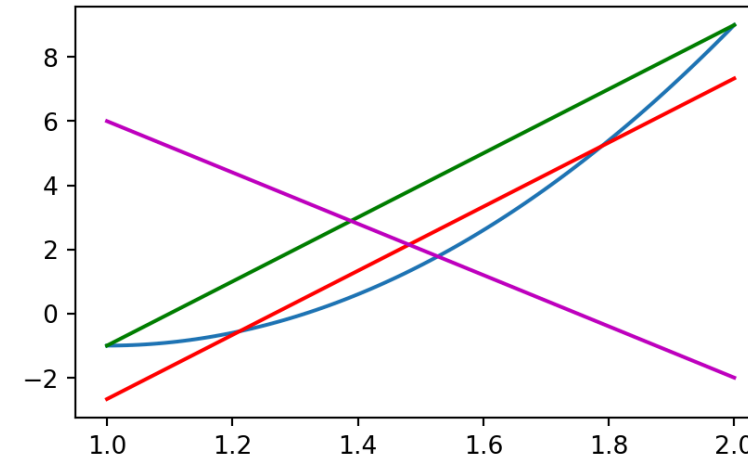
How to find the **best**  $\{\hat{u}_0, \hat{u}_1\}$ ?

# How to find the best approximation?

What options are there? Who decides what is **best**?

The three methods covered in this class are

- The least squares method
- The Galerkin method
- The collocation method



The first two are **variational** methods, whereas the third is an **interpolation** method.

Each method will give us  $N + 1$  equations for the  $N + 1$  unknowns  $\{\hat{u}_k\}_{k=0}^N$ !

- A variational method derives equations by using integration over the domain
- A collocation method derives equations at different points in the mesh

# Variational methods

Since a variational method derives equations by integrating over the domain, we need to define a special notation. The  $L^2$  inner product is defined as

$$(f, g) = \int_{\Omega} f(x)g(x)d\Omega,$$

for two real functions  $f(x)$  and  $g(x)$  (complex functions have a slightly different inner product). The symbol  $\Omega$  here represents the domain of interest. For our first example  $\Omega = [1, 2]$ .

## Note

Sometimes the inner product is written as  $(f, g)_{L^2(\Omega)}$ , in order to clarify that it is the  $L^2$  inner product on a certain domain. Normally, the  $L^2(\Omega)$  subscript will be dropped.

# The $L^2(\Omega)$ norm

The inner product is used to define an  $L^2(\Omega)$  norm

$$\|u\| = \sqrt{(u, u)}$$

The norm gives us a measure for the length or size of a vector.

## The $L^2(\Omega)$ error norm

If we define the pointwise error as

$$e(x) = u(x) - u_N(x)$$

then the  $L^2(\Omega)$  error norm

$$\|e\| = \sqrt{(e, e)} = \sqrt{\int_{\Omega} e^2 dx}$$

gives us a measure of the error over the entire domain



# Back to our example - how do we find the best possible $u_N \approx u$ ?

If we use the error

$$e(x) = u(x) - u_N(x)$$

then the  $L^2$  error norm  $\|e\|$  represents the error integrated over the entire domain.

 Idea!

Why don't we find  $u_N$  such that it minimizes the  $L^2$  error norm!

→ **The Least squares method!**

# The least squares method for function approximation

We want to approximate

$$u_N(x) \approx u(x)$$

using a function space  $V_N$  for  $u_N$ . The pointwise error in  $u_N$  is defined as

$$e(x) = u(x) - u_N(x)$$

Define the square of the  $L^2$  error norm

$$E = \|e\|^2 = (e, e)$$

The least squares method is to find  $u_N \in V_N$  such that

$$\frac{\partial E}{\partial \hat{u}_j} = 0, \quad j = 0, 1, \dots, N.$$

# The least squares method

Find  $u_N \in V_N$  such that

$$\frac{\partial E}{\partial \hat{u}_j} = 0, \quad j = 0, 1, \dots, N$$

This gives us  $N + 1$  equations for  $N + 1$  unknowns  $\{\hat{u}_j\}_{j=0}^N$ .

But how can we compute  $\frac{\partial E}{\partial \hat{u}_j}$ ?

$$\frac{\partial E}{\partial \hat{u}_j} = \frac{\partial}{\partial \hat{u}_j} \int_{\Omega} e^2 dx = \int_{\Omega} 2e \frac{\partial e}{\partial \hat{u}_j} dx$$

Insert for  $e(x) = u(x) - u_N(x) = u(x) - \sum_{k=0}^N \hat{u}_k \psi_k$

$$\frac{\partial E}{\partial \hat{u}_j} = \int 2 \left( u - \sum_{k=0}^N \hat{u}_k \psi_k \right) \overbrace{\frac{\partial}{\partial \hat{u}_j} \left( u - \sum_{k=0}^N \hat{u}_k \psi_k \right)}^{-\psi_j} dx$$

If we now set  $\frac{\partial E}{\partial \hat{u}_j} = 0$ , then

$$\frac{\partial E}{\partial \hat{u}_j} = - \int 2 \left( u - \sum_{k=0}^N \hat{u}_k \psi_k \right) \psi_j dx = 0$$

$$\longrightarrow \int u \psi_j dx = \sum_{k=0}^N \int \psi_k \psi_j dx \hat{u}_k, \quad j = 0, 1, \dots, N$$

gives us  $N + 1$  linear equations for the  $N + 1$  unknown  $\{\hat{u}_k\}_{k=0}^N$

**Back to the example:**  $u(x) = 10(x - 1)^2 - 1$  for  $x \in [1, 2]$

Find  $u_N \in \mathbb{P}_1$  (meaning find  $\hat{u}_0$  and  $\hat{u}_1$ ) such that

$$\frac{\partial E}{\partial \hat{u}_0} = 0 \text{ and } \frac{\partial E}{\partial \hat{u}_1} = 0$$

Two equations for two unknowns.

# Implement the 2 equations in Sympy

```
1 def inner(u, v, domain=(-1, 1), x=x):
2     return sp.integrate(u*v, (x, domain[0], domain[1]))
3
4 u0, u1 = sp.symbols('u0,u1')
5 err = u-(u0+u1*x)
6 E = inner(err, err, (1, 2))
7 eq1 = sp.Eq(sp.diff(E, u0, 1), 0)
8 eq2 = sp.Eq(sp.diff(E, u1, 1), 0)
9 display(eq1)
10 display(eq2)
```

$$2u_0 + 3u_1 - \frac{14}{3} = 0$$

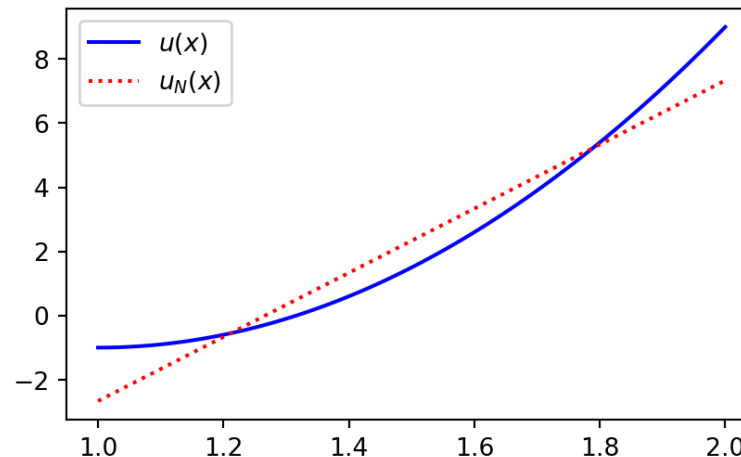
$$3u_0 + \frac{14u_1}{3} - \frac{26}{3} = 0$$

```
1 uhat = sp.solve((eq1, eq2), (u0, u1))
2 print(uhat)
```

{u0: -38/3, u1: 10}

The LSM best fit equation is

$$u_N = -38/3 + 10x$$



# Linear algebra approach

$$\underbrace{\int u \psi_j dx}_{b_j} = \sum_{k=0}^N \underbrace{\int \psi_k \psi_j dx}_{a_{jk}} \underbrace{\hat{u}_k}_{x_k}, \quad j = 0, 1, \dots, N.$$

$$\mathbf{b} = \mathbf{A}\mathbf{x} \longrightarrow \mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

Assemble mass matrix  $a_{jk} = \int \psi_k \psi_j dx$  and vector  $b_j = \int u \psi_j dx$

```
1 A = sp.zeros(2, 2)
2 b = sp.zeros(2, 1)
3 psi = (sp.S(1), x)
4 u = 10*(x-1)**2-1
5 for i in range(2):
6     for j in range(2):
7         A[i, j] = sp.integrate(psi[i]*psi[j], (x, 1, 2))
8     b[i] = sp.integrate(psi[i]*u, (x, 1, 2))
9 print(A)
10 print(A**(-1) @ b)
```

```
Matrix([[1, 3/2], [3/2, 7/3]])
Matrix([[ -38/3], [10]])
```

Same result as before!

# The Galerkin method is a more popular variational method for solving differential equations

Here the error is required to be orthogonal to the basis functions (for the  $L^2$  inner product)

$$(e, \psi_j) = 0, \quad j = 0, 1, \dots, N.$$

Again  $N + 1$  equations for  $N + 1$  unknowns.

Not as much work as the least squares method, and even easier to implement in SymPy:

```
1 eq1 = sp.Eq(inner(err, 1, domain=(1, 2)), 0)
2 eq2 = sp.Eq(inner(err, x, domain=(1, 2)), 0)
3 display(eq1)
4 display(eq2)
```

$$-u_0 - \frac{3u_1}{2} + \frac{7}{3} = 0$$
$$-\frac{3u_0}{2} - \frac{7u_1}{3} + \frac{13}{3} = 0$$

```
1 sp.solve((eq1, eq2), (u0, u1))
```

```
{u0: -38/3, u1: 10}
```

# The Galerkin formulation of the problem is

Find  $u_N \in V_N$  such that

$$(e, v) = 0, \quad \forall v \in V_N$$

Here  $v$  is often referred to as a **test** function, whereas  $u_N$  is referred to as a **trial** function.

Alternative formulation:

Find  $u_N \in V_N$  such that

$$(e, \psi_j) = 0, \quad j = 0, 1, \dots, N.$$

## Note

In order to satisfy  $(e, v) = 0$  for all  $v \in V_N$ , we can insert for  $v = \sum_{j=0}^N \hat{v}_j \psi_j$  such that

$$\sum_{j=0}^N (e, \psi_j) \hat{v}_j = 0.$$

In order for this to always be satisfied, we require that  $(e, \psi_j) = 0$  for all  $j = 0, 1, \dots, N$ .



# Linear algebra problem for the Galerkin method

We have the  $N + 1$  equations

$$(e, \psi_i) = 0, \quad i = 0, 1, \dots, N.$$

Insert for  $e = u - u_N = u - \sum_{j=0}^N \hat{u}_j \psi_j$  to get

$$(u - \sum_{j=0}^N \hat{u}_j \psi_j, \psi_i) = 0, \quad i = 0, 1, \dots, N$$

and thus the linear algebra problem  $A\mathbf{x} = \mathbf{b}$ :

$$\sum_{j=0}^N \underbrace{(\psi_j, \psi_i)}_{a_{ij}} \underbrace{\hat{u}_j}_{x_j} = \underbrace{(u, \psi_i)}_{b_i}, \quad i = 0, 1, \dots, N$$

# Orthogonality is a big deal!

If we choose basis  $\{\psi_j\}_{j=0}^N$  such that all basis functions are **orthonormal**, then

$$a_{ij} = (\psi_j, \psi_i) = \delta_{ij} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$$

and thus

$$A\mathbf{x} = \mathbf{b} \rightarrow \mathbf{x} = \mathbf{b} \quad \text{or} \quad \hat{u}_j = (u, \psi_j), \quad j = 0, 1, \dots, N$$

If the basis functions are merely orthogonal, then

$$(\psi_j, \psi_i) = \|\psi_i\|^2 \delta_{ij},$$

where  $\|\psi_i\|^2$  is the squared  $L^2$  norm of  $\psi_i$ . We can still easily solve the linear algebra system (because  $A$  is a diagonal matrix)

$$\hat{u}_i = \frac{(u, \psi_i)}{\|\psi_i\|^2} \quad \text{for } i = 0, 1, \dots, N$$

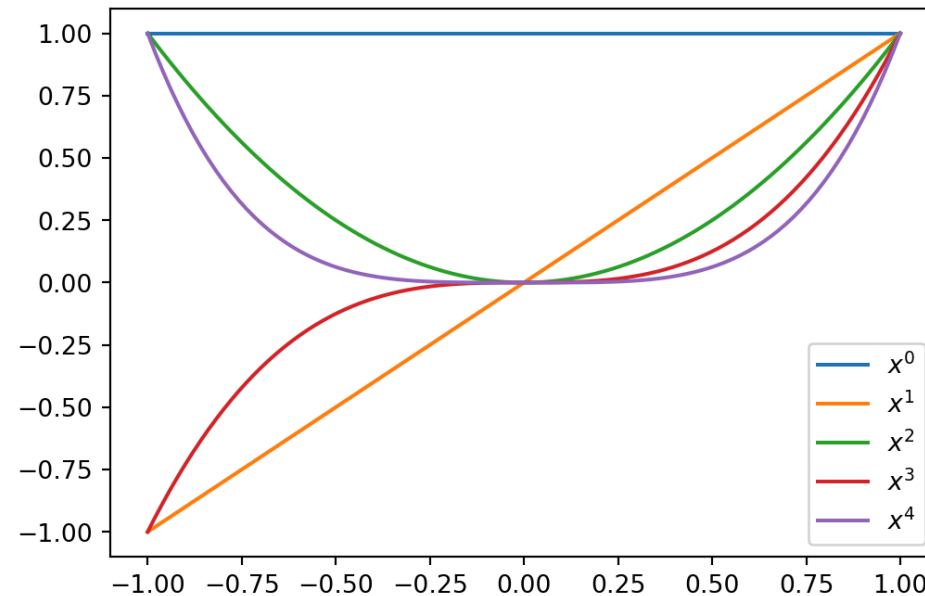
# Polynomial basis functions

The monomial basis

$$\{x^n\}_{n=0}^N,$$

is a basis for  $\mathbb{P}_N$ . However, it is not a good basis. This is because the basis functions are **not orthogonal** and the mass matrix  $A$  is **ill conditioned**. In short, this means it is difficult to solve  $A\mathbf{x} = \mathbf{b}$  with good accuracy using finite precision computers.

The basis functions are shown below for  $x \in [-1, 1]$

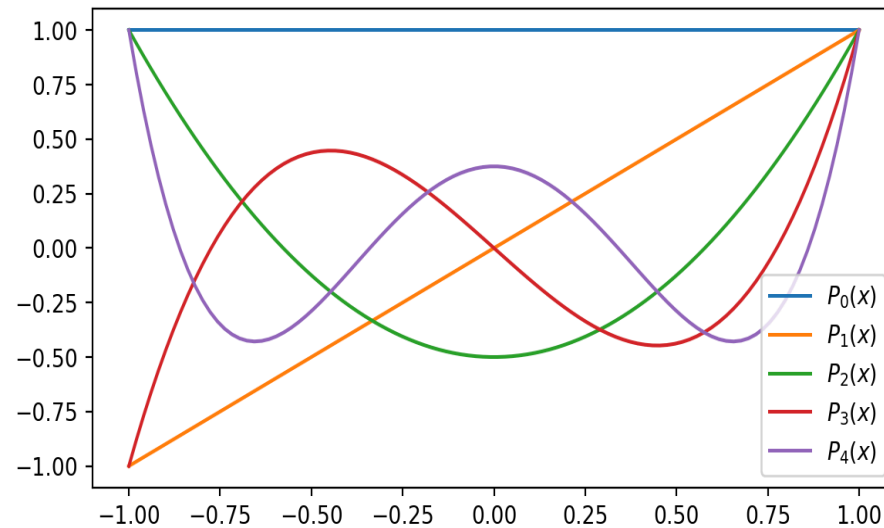


# Legendre polynomials are much better basis functions

Legendre polynomials,  $P_i(x)$ , are defined on the domain  $\Omega = [-1, 1]$  as the recursion

$$\begin{aligned}P_0(x) &= 1, \\P_1(x) &= x, \\P_2(x) &= \frac{1}{2}(3x^2 - 1), \\&\vdots \\(j+1)P_{j+1}(x) &= (2j+1)xP_j(x) - jP_{j-1}(x).\end{aligned}$$

The first 5 Legendre polynomials are plotted on the right



# The Legendre polynomials are $L^2$ orthogonal on the reference domain $\Omega = [-1, 1]$

$$(P_j, P_i)_{L^2(-1,1)} = \frac{2}{2j+1} \delta_{ij}$$

Orthogonality in the  $L^2$  inner product space makes the Legendre polynomials very popular!

Using  $\|P_j\|^2 = \frac{2}{2j+1}$  we get that

$$\hat{u}_j = \frac{(u, P_j)}{\|P_j\|^2} = \frac{2j+1}{2} (u, P_j)$$

## Note

This requires that the domain of  $u(x)$  is  $[-1, 1]$

# If the physical domain $[a, b]$ is different from the reference domain $[-1, 1]$ , then we need to map

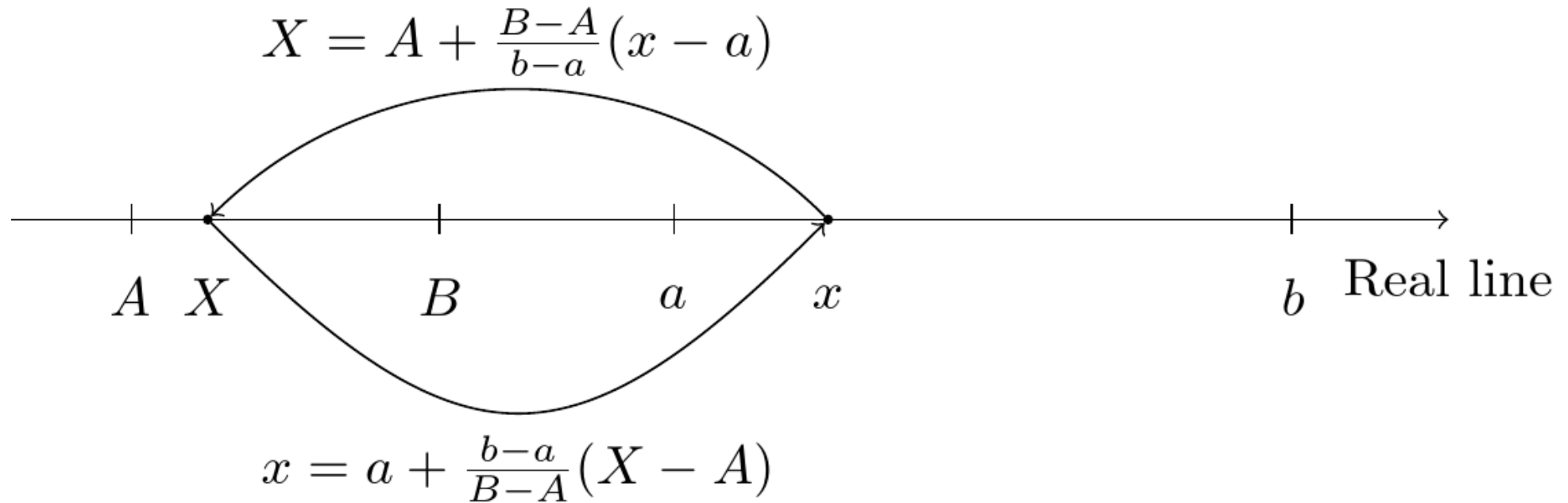
Many well-known basis functions work only on a given reference domain

- Sines and cosines  $[0, \pi]$
- Legendre polynomials  $[-1, 1]$
- Bernstein polynomials  $[0, 1]$
- Chebyshev polynomials  $[-1, 1]$

Let  $X$  be the coordinate in the computational (reference) domain  $[A, B]$  and  $x$  be the coordinate in the true physical domain  $[a, b]$ . A linear (affine) mapping, from  $X$  to  $x$  (and back) is then

$$X \in [A, B] \quad \text{and} \quad x \in [a, b]$$
$$x = a + \frac{b - a}{B - A}(X - A) \quad \text{and} \quad X = A + \frac{B - A}{b - a}(x - a)$$

# Affine map



- Equations are defined in the real domain  $[a, b]$
- Equations are solved in the computational domain  $[A, B]$

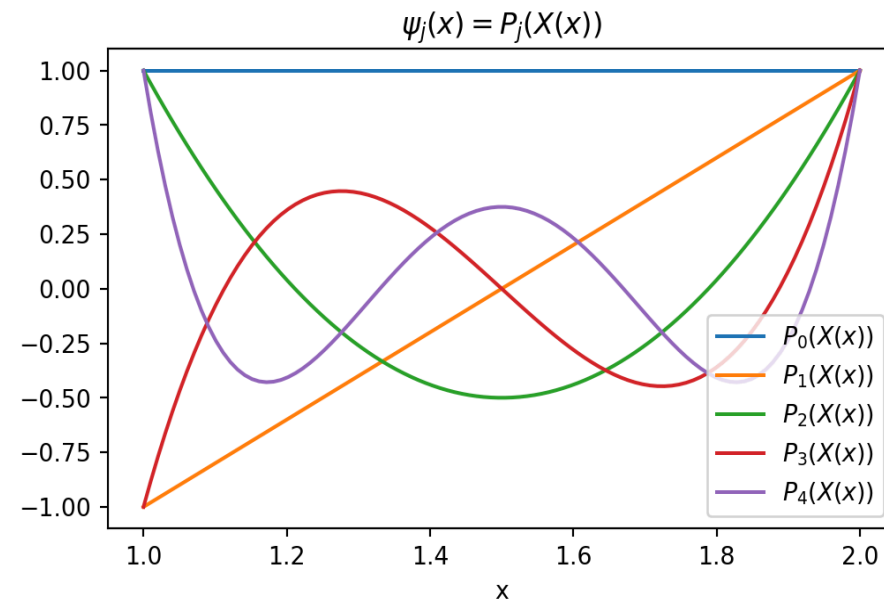
# The mapping makes it possible to use Legendre polynomials in any domain $[a, b]$

The basis functions  $\psi_j(x)$  are simply

$$\psi_j(x) = P_j(X(x)) \quad X \in [-1, 1], \quad x \in [a, b]$$

$$X(x) = -1 + \frac{2}{b-a}(x-a)$$

```
1 xj = np.linspace(1, 2, 100)
2 A, B = -1, 1
3 a, b = 1, 2
4 Xj = -1 + (B-A)/(b-a)*(xj-a)
5 plt.figure(figsize=(6, 3.5))
6 legend = []
7 p = np.zeros(100)
8 for n in range(5):
9     l = sp.legendre(n, x)
10    p[:] = sp.lambdify(x, l)(Xj)
11    plt.plot(xj, p)
12    legend.append(f'$P_{n}(X(x))$')
13 plt.title(r'$\psi_j(x)=P_j(X(x))$')
14 plt.xlabel('x')
15 plt.legend(legend);
```





# The mapping complicates the inner product

The Galerkin method is defined on the true domain using  $L^2([a, b])$

$$(u(x) - u_N(x), \psi_j(x))_{L^2([a, b])} = \int_a^b (u(x) - u_N(x)) \psi_j(x) dx = 0, \quad j = 0, 1, \dots, N.$$

Insert for  $u_N = \sum_{k=0}^N \hat{u}_k \psi_k(x)$  and rearrange

$$\sum_{k=0}^N \int_a^b \psi_k(x) \psi_j(x) dx \hat{u}_k = \int_a^b u(x) \psi_j(x) dx, \quad j = 0, 1, \dots, N.$$

So far just regular Galerkin.

Now introduce  $\psi_j(x) = P_j(X(x))$  and  $\psi_k(x) = P_k(X(x))$  integrate with a change of variables  $x \rightarrow X$ . The new integration limits are then  $X(a) = -1$  and  $X(b) = 1$

$$\sum_{k=0}^N \int_{-1}^1 P_k(X) P_j(X) \cancel{\frac{dx}{dX}} dX \hat{u}_k = \int_{-1}^1 u(x(X)) P_j(X) \cancel{\frac{dx}{dX}} dX, \quad j = 0, 1, \dots, N$$

**We get the linear system with inner products over the reference domain**

$$\sum_{k=0}^N \int_{-1}^1 P_k(X) P_j(X) dX \hat{u}_k = \int_{-1}^1 u(x(X)) P_j(X) dX, \quad j = 0, 1, \dots, N$$

which can be written as

$$\sum_{k=0}^N \underbrace{(P_k(X), P_j(X))_{L^2(-1,1)}}_{\frac{2}{2j+1} \delta_{kj}} \hat{u}_k = (u(x(X)), P_j(X))_{L^2(-1,1)}$$

such that

$$\hat{u}_j = \frac{2j+1}{2} (u(x(X)), P_j(X))_{L^2(-1,1)} \quad j = 0, 1, \dots, N$$

# Implementation of the mapped inner product

The following inner product is valid for any mapping

```
1 def inner(u, v, domain, ref_domain=(-1, 1)):
2     A, B = ref_domain
3     a, b = domain
4     X = a + (b-a)*(x-A)/(B-A)
5     us = u.subs(x, X) # u(x(X))
6     return sp.integrate(us*v, (x, A, B))
```

For example, for our  $u(x) = 10(x - 1)^2 - 1$  in the domain  $\Omega = [a, b] = [1, 2]$  we get

```
1 uhat = lambda u, j: (2*j+1) * inner(u, sp.legendre(j, x), (1, 2))/2
2 u = 10*(x-1)**2-1
3 u0, u1 = uhat(u, 0), uhat(u, 1)
4 print(u0, u1)
5 print(f"uN(x) = {u0}+{u1}X(x)")
```

7/3 5

$u_N(x) = 7/3 + 5X(x)$

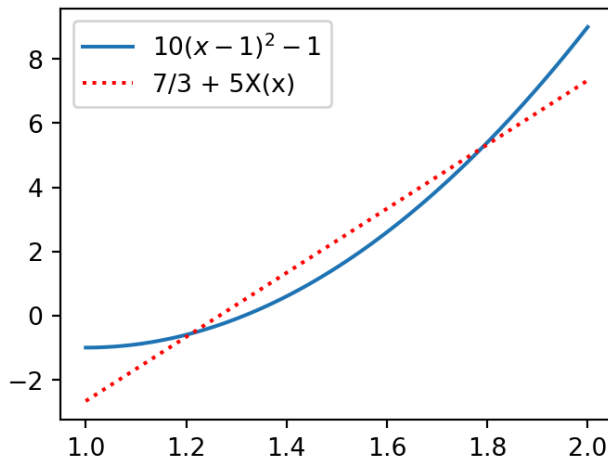
The Legendre polynomials use the reference coordinate  $X$ , whereas the true function  $u_N(x)$  is a function of  $x$  from the true space.

## Note

The `inner` product implemented here is using Sympy's `integrate` function, which may be too slow or not find the solution at all. It is normally better to use `scipy.integrate.quad`

# Plot the Legendre approximation

```
1 plt.figure(figsize=(4, 3))
2 xj = np.linspace(1, 2, 100)
3 Xj = -1 + 2/(b-a)*(xj-a)
4 plt.plot(xj, sp.lambdify(x, u)(xj))
5 plt.plot(xj, uhat(u, 0) + uhat(u, 1)*Xj, 'r:')
6 plt.legend(['$10(x-1)^2-1$', f'{uhat(u, 0)} + {uhat(u, 1)}X(x)']);
```



The result is exactly the same as was found with the monomial basis  $\{1, x\}$ .



Tip

Numpy has a complete module dedicated to [Legendre](#) polynomials

# Summary of variational methods

We want to approximate  $u(x)$  with  $u_N(x) \in V_N$

$$u_N(x) \approx u(x)$$

using the inner product

$$(f, g) = \int_a^b f(x)g(x)dx$$

## Least squares method

Find  $u_N \in V_N$  such that

$$\frac{\partial E}{\partial \hat{u}_j} = 0, \quad j = 0, 1, \dots, N$$

where  $E = (u - u_N, u - u_N)$ .

Minimizing the  $L^2$  error norm.

## Galerkin method

Find  $u_N \in V_N$  such that

$$(u - u_N, v) = 0, \quad \forall v \in V_N$$

Or equivalently

$$(u - u_N, \psi_j) = 0, \quad j = 0, 1, \dots, N$$

Error orthogonal to the test functions

# The collocation method

Takes a very different approach than the variational methods, but the objective is still to find  $u_N \in V_N$  such that

$$u_N(x) = \sum_{j=0}^N \hat{u}_j \psi_j(x)$$

The collocation method requires that for some  $N + 1$  **chosen** mesh points  $\{x_j\}_{j=0}^N$  the following  $N + 1$  equations are satisfied

$$u(x_i) - u_N(x_i) = 0, \quad i = 0, 1, \dots, N.$$

Inserting for  $u_N(x_i)$  we get the  $N + 1$  linear algebra equations

$$\sum_{j=0}^N \hat{u}_j \underbrace{\psi_j(x_i)}_{a_{ij}} = u(x_i) \longrightarrow \sum_{j=0}^N a_{ij} \hat{u}_j = u(x_i), \quad i = 0, 1, \dots, N$$

# The Lagrange interpolation method described in **lecture 7** is a collocation method

The Lagrange basis functions  $\ell_j(x)$  are defined for points  $\{x_j\}_{j=0}^N$  such that

$$\ell_j(x) = \prod_{\substack{0 \leq m \leq N \\ m \neq j}} \frac{x - x_m}{x_j - x_m} \quad \text{and} \quad \ell_j(x_i) = \delta_{ij} = \begin{cases} 1 & \text{for } i = j \\ 0 & \text{for } i \neq j \end{cases}$$

Hence the matrix  $a_{ij} = \psi_j(x_i) = \delta_{ij}$  and we can simply use the coefficients

$$\hat{u}_j = u(x_j) \quad \text{such that} \quad u_N(x) = \sum_{j=0}^N u(x_j) \ell_j(x)$$

There is no integration and the method is often favored for its simplicity. There is a problem though. How do you choose the collocation points?!

## Note

The Lagrange polynomial here is using all  $N + 1$  mesh points. This is different from lecture 7, where we only used a few mesh points close to the interpolation point.

# Lagrange collocation method for

$$u(x) = 10(x - 1)^2 - 1, x \in [1, 2]$$

The approximation using two collocation points (linear function,  $u_N \in V_N = \text{span}\{1, x\}$ ) is

$$u_N(x) = \hat{u}_0 \ell_0(x) + \hat{u}_1 \ell_1(x),$$

or more simply

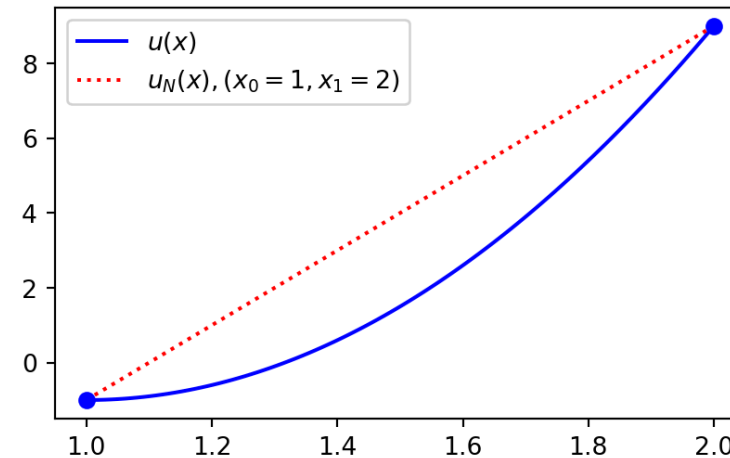
$$u_N(x) = u(x_0) \ell_0(x) + u(x_1) \ell_1(x).$$

We can choose the end points  $x_0 = 1$  and  $x_1 = 2$  and reuse the two functions [Lagrangebasis](#) and [Lagrangefunction](#) from [lecture 7](#). The result is then as shown on the next slide



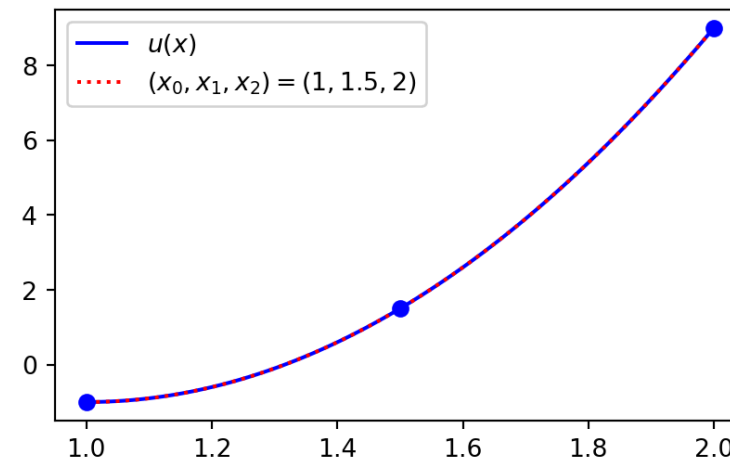
# Collocation

```
1 from lagrange import Lagrangebasis, Lagrangefunction
2 xj = np.linspace(1, 2, 100)
3 u = 10*(x-1)**2-1
4 xp = np.array([1, 2])
5 ell = Lagrangebasis(xp)
6 L = Lagrangefunction([u.subs(x, xi) for xi in xp],
7 plt.figure(figsize=(5, 3))
8 plt.plot(xj, sp.lambdify(x, u)(xj), 'b')
9 plt.plot(xj, sp.lambdify(x, L)(xj), 'r:')
10 plt.plot(xp, [u.subs(x, xi) for xi in xp], 'bo')
11 plt.legend(['$u(x)$', '$u_N(x), (x_0=1, x_1=2)$']];
```



Use three points and the approximation is perfect because  $u(x)$  is a 2nd order polynomial

```
1 xp = np.array([1, 1.5, 2])
2 ell = Lagrangebasis(xp)
3 L = Lagrangefunction([u.subs(x, xi) for xi in xp],
4 plt.figure(figsize=(5, 3))
5 plt.plot(xj, sp.lambdify(x, u)(xj), 'b')
6 plt.plot(xj, sp.lambdify(x, L)(xj), 'r:')
7 plt.plot(xp, [u.subs(x, xi) for xi in xp], 'bo')
8 plt.legend(['$u(x)$', '$(x_0, x_1, x_2) = (1, 1.5, 2)$']];
```



# High order interpolation on uniform grids is bad

Lets consider a more difficult function

$$u(x) = \frac{1}{1 + 25x^2}, \quad x \in [-1, 1],$$

and attempt to approximate it with Lagrange polynomials on a uniform grid.

Use

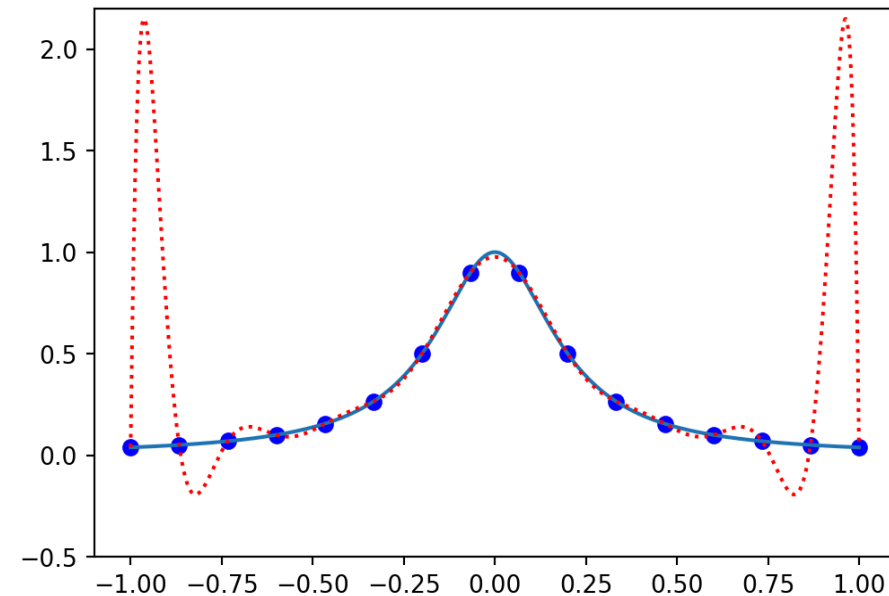
$$N = 15$$

$$x_i = -1 + \frac{2i}{N}, \quad i = 0, 1, \dots, N$$

$$u(x_i) = u_N(x_i) = \sum_{j=0}^N u(x_j) \ell_j(x_i), \quad i = 0, 1, \dots, N$$

# Implementation

```
1 u = 1/(1+25*x**2)
2 N = 15
3 xj = np.linspace(-1, 1, N+1)
4 uj = sp.lambdify(x, u)(xj)
5 ell = Lagrangebasis(xj)
6 L = Lagrangefunction(uj, ell)
7 yj = np.linspace(-1, 1, 1000)
8 plt.figure(figsize=(6, 4))
9 plt.plot(xj, uj, 'bo')
10 plt.plot(yj, sp.lambdify(x, u)(yj))
11 plt.plot(yj, sp.lambdify(x, L)(yj), 'r:')
12 ax = plt.gca()
13 ax.set_ylim(-0.5, 2.2);
```



Large over and undershoots in the approximation Lagrange polynomial!

Runge's phenomenon! Interpolation on uniform grids may lead to large over and undershoots.

Why?

# Runge's phenomenon

Define the monic polynomial with roots in  $\{x_j\}_{j=0}^N$  as

$$p_N(x) = \prod_{j=0}^N (x - x_j)$$

It can be shown that the error in the Lagrange polynomial  $u_N(x)$  using  $\{x_j\}_{j=0}^N$  is

$$u(x) - u_N(x) = \frac{u^{(N)}(\xi)}{(N+1)!} p_N(x)$$

for some  $\xi \in [a, b]$ .

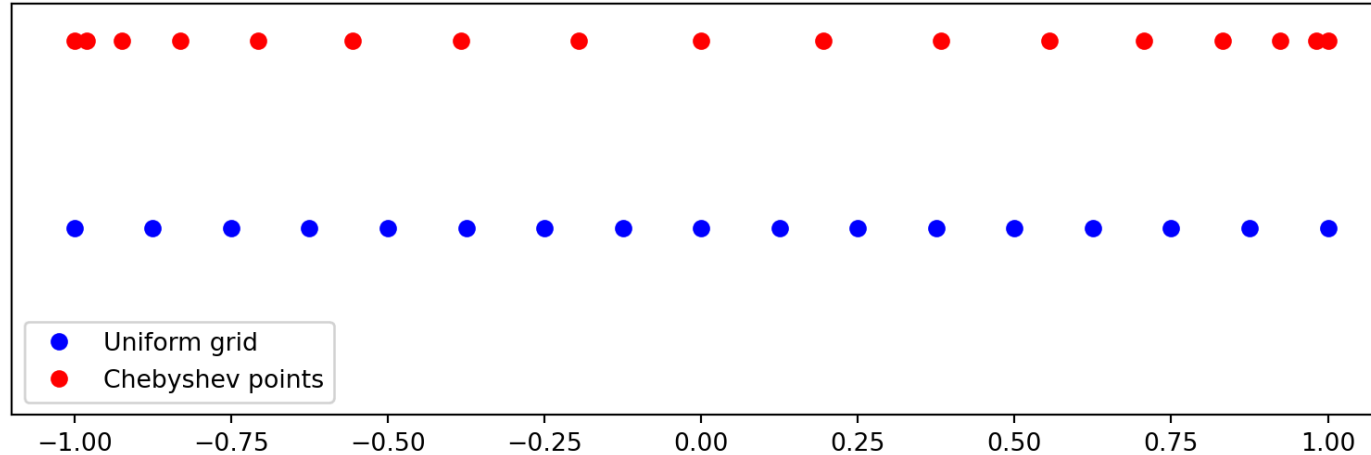
Hence, large errors occur when  $p_N(x)$  is large or  $u^{(N)}(\xi) = \frac{d^N u}{dx^N}(\xi)$  is large.

Lets look at  $p_N(x)$ , which is independent of the function  $u(x)$ !

# The monic $p_N(x)$ can be created for any mesh

Lets use both a uniform mesh and Chebyshev points:

$$x_j = -1 + \frac{2j}{N} \quad \text{and} \quad x_j = \cos(j\pi/N), \quad j = 0, 1, \dots, N$$

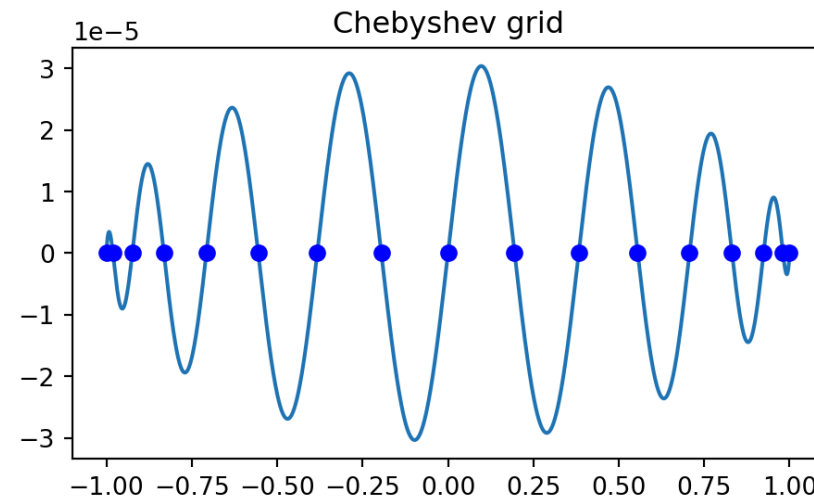
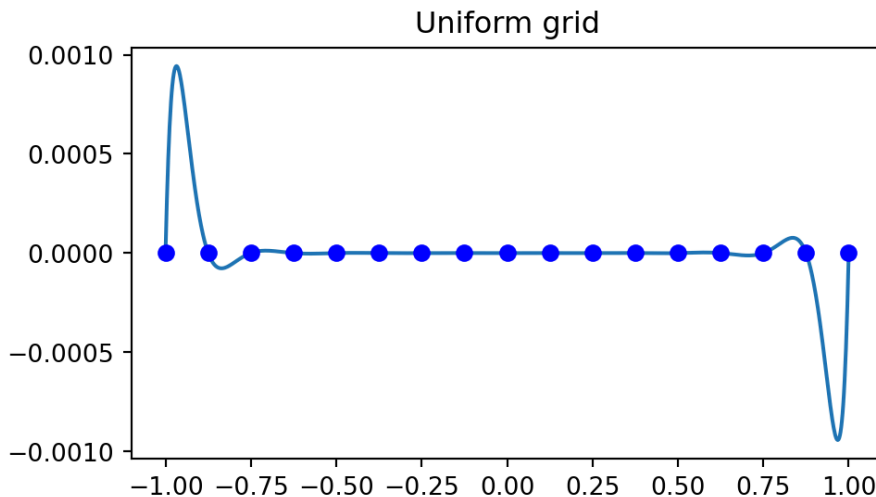


We see that Chebyshev points are clustered near the edges

Lets construct  $p_N(x)$  using either the uniform mesh or Chebyshev nodes..

# Plot the monic polynomial $p_N(x) = \prod_{j=0}^N (x - x_j)$ at uniform and Chebyshev mesh

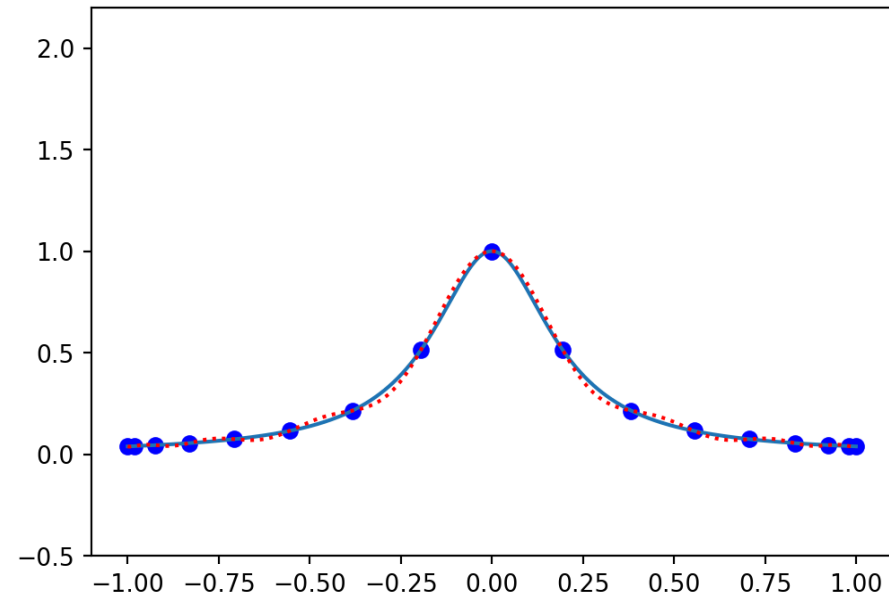
```
1 xp = np.linspace(-1, 1, N+1); xc = np.cos(np.arange(N+1)*np.pi/N)
2 pp = np.poly(xp); pc = np.poly(xc)
3 xj = np.linspace(-1, 1, 1000)
4 fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 3))
5 ax1.plot(xj, np.polyval(pp, xj)); ax2.plot(xj, np.polyval(pc, xj))
6 ax1.plot(xp, np.zeros(N+1), 'bo'); ax2.plot(xc, np.zeros(N+1), 'bo')
7 ax1.set_title('Uniform grid'); ax2.set_title('Chebyshev grid');
```



- A uniform mesh leads to large oscillations near the edges, whereas Chebyshev points leads to a polynomial with near uniform oscillations.
- The oscillations on the uniform mesh grow even larger when increasing  $N$ .

# Compute the Lagrange polynomial using Chebyshev points

```
1 xj = np.cos(np.arange(N+1)*np.pi/N)
2 uj = sp.lambdify(x, u)(xj)
3 ell = Lagrangebasis(xj)
4 L = Lagrangefunction(uj, ell)
5 plt.figure(figsize=(6, 4))
6 plt.plot(xj, uj, 'bo')
7 plt.plot(yj, sp.lambdify(x, u)(yj))
8 plt.gca().set_ylim(-0.5, 2.2);
9 plt.plot(yj, sp.lambdify(x, L)(yj), 'r:');
```



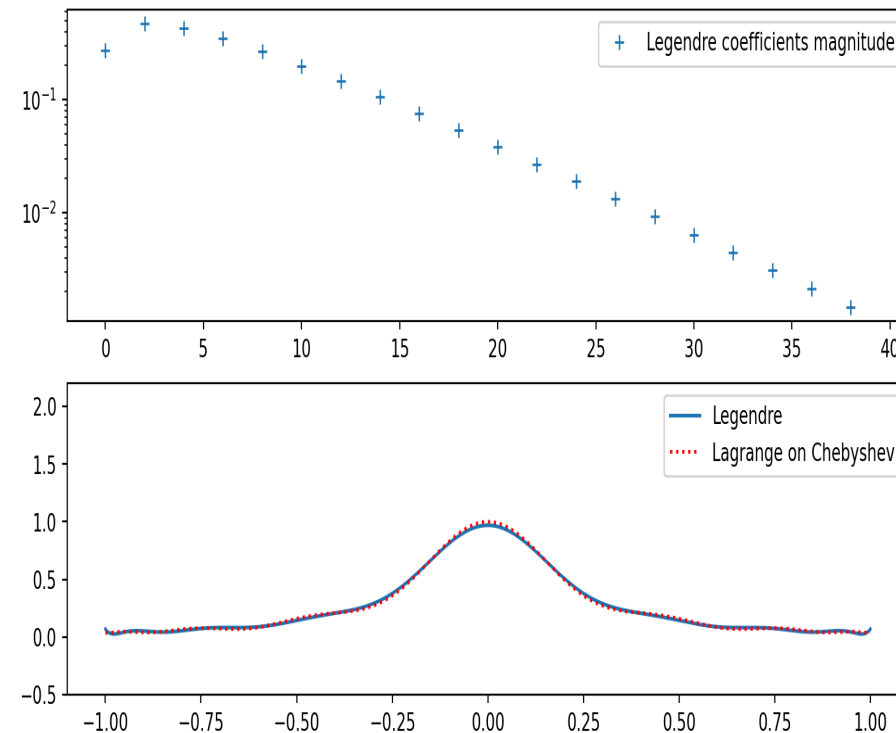
Only small oscillations - interpolation is converging using more points!

# The variational method and $u(x) = \frac{1}{1+25x^2}$

Lets try Legendre polynomials. There is no need for mapping since the domain is  $[-1, 1]$ . And there are no mesh points! The Legendre coefficients are

$$\hat{u}_j = \frac{2j+1}{2}(u, P_j), \quad j = 0, 1, \dots$$

```
1 # Use scipy to compute integral
2 from numpy.polynomial import Legendre
3 from scipy.integrate import quad
4 def innern(u, v):
5     uj = lambda xj: sp.lambdify(x, u)(xj)*v(xj)
6     return quad(uj, -1, 1)[0]
7
8 u = 1/(1+25*x**2)
9 uhat = lambda u, j: (2*j+1) * innern(u, Legendre.basis(j))
10 ul = [uhat(u, n) for n in range(40)]
11
12 xj = np.linspace(-1, 1, 1000)
13 uj = sp.lambdify(x, u)(xj)
14 fig, (ax1, ax2) = plt.subplots(2, 1)
15 ax1.semilogy(abs(np.array(ul)), '+')
16 ax2.plot(xj, Legendre(ul[:17])(xj))
17 ax2.plot(xj, sp.lambdify(x, L)(xj), 'r:')
18 ax2.set_ylim(-0.5, 2.2);
19 ax1.legend(['Legendre coefficients magnitude'])
20 ax2.legend(['Legendre', 'Lagrange on Chebyshev'])
```





# Error in Legendre approximations

The smooth function  $u(x)$  can be exactly represented as

$$u(x) = \sum_{k=0}^{\infty} \hat{u}_k P_k(x)$$

For all  $k \leq N$  the series for  $u(x)$  and the series for  $u_N(x)$  have exactly the same coefficients  $\hat{u}_k$ . Hence the error in the approximation  $u_N$  is easily computed as

$$e(x) = u(x) - u_N(x) = \sum_{k=N+1}^{\infty} \hat{u}_k P_k(x)$$

Spectral convergence - it can be shown that

$$\|u - u_N\| \sim |\hat{u}_{N+1}| \sim e^{-\mu N}, \quad \mu \in \mathbb{R}^+$$

Exponential convergence!

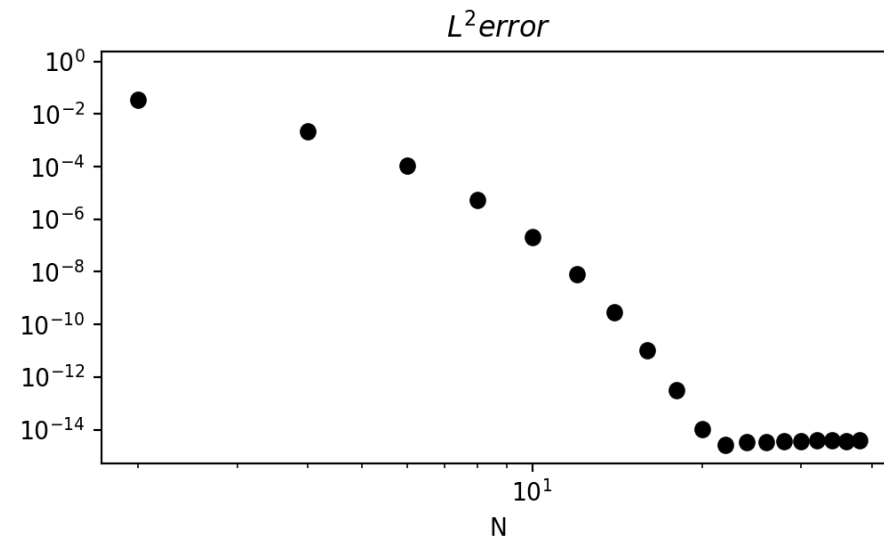
# An example of spectral (exponential) convergence

Lets apprximate the smooth function

$$u = e^{\cos(x)}, \quad x \in [-1, 1]$$

and monitor the  $L^2$  error as we increase  $N$ .

```
1 def L2_error(uh, ue):
2     uej = sp.lambdify(x, ue)
3     err = []
4     for n in range(0, 40, 2):
5         uf = lambda xj: (Legendre(uh[:n+1]))(xj)
6                     -uej(xj))**2
7         err.append(np.sqrt(quad(uf, -1, 1)[0]))
8     return err
9 u = sp.exp(sp.cos(x))
10 # Compute Legendre coefficients
11 uh = [uhat(u, n) for n in range(0, 40)]
12 plt.figure(figsize=(6, 3))
13 plt.loglog(np.arange(0, 40, 2), L2_error(uh, u), 'k')
14 plt.title('$L^2$ error$');plt.xlabel('N');
```



- Exponentially fast convergence rate!
- Error does not go down further due to machine precision.

# Boundary issues

Lets consider a slightly different problem

$$u(x) = 10(x - 1)^2 - 1, \quad x \in [0, 1]$$

and attempt to find  $u_N \in V_N = \text{span}\{\sin((j + 1)\pi x)\}_{j=0}^N$  with the Galerkin method.

The sines are orthogonal such that the mass matrix becomes diagonal

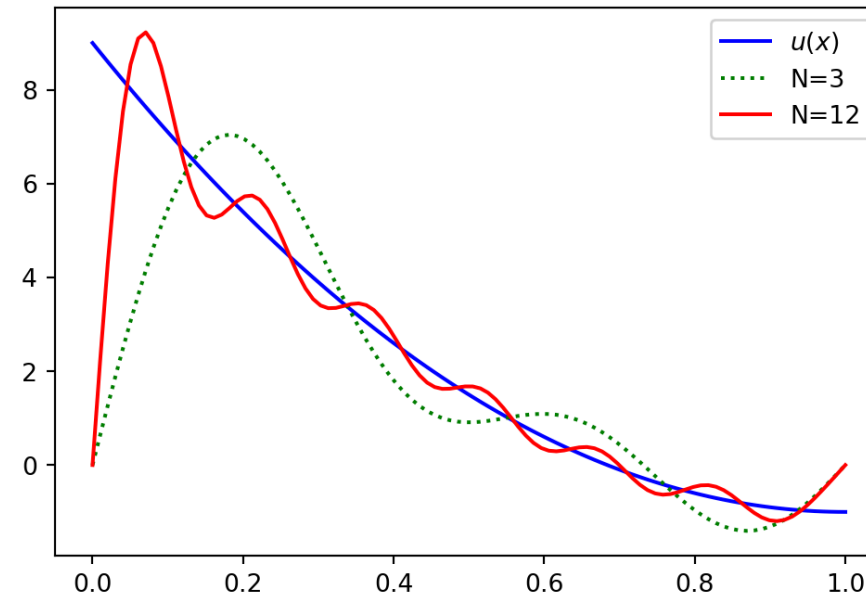
$$\int_0^1 \sin((j + 1)\pi x) \sin((i + 1)\pi x) dx = \frac{1}{2} \delta_{ij}$$

Hence we can easily get the coefficients with the Galerkin method

$$\hat{u}_i = 2(u(x), \sin((i + 1)\pi x))_{L^2(0,1)}$$

# Implementation

```
1 u = 10*(x-1)**2-1
2 uhat = lambda u, i: 2*inner(u, sp.sin((i+1)*sp.pi*x))
3 ul = []
4 for i in range(15):
5     ul.append(uhat(u, i).n())
6
7 ul = np.array(ul, dtype=float)
8 def uN(uh, xj):
9     N = len(xj)
10    uj = np.zeros(N)
11    for i, ui in enumerate(uh):
12        uj[:] += ui*np.sin((i+1)*np.pi*xj)
13    return uj
14
15 xj = np.linspace(0, 1, 100)
16 plt.figure(figsize=(6,4))
17 plt.plot(xj, 10*(xj-1)**2-1, 'b')
18 plt.plot(xj, uN(ul[:3+1]), xj), 'g:')
19 plt.plot(xj, uN(ul[:12+1]), xj), 'r-')
20 plt.legend(['$u(x)$', 'N=3', 'N=12']);
```



What is wrong?

All basis functions are zero at the domain boundaries! And as such

$$u_N(0) = u_N(1) = 0$$

# Solution, add a boundary function with nonzero boundary values

$$u_N(x) = B(x) + \tilde{u}_N(x) = B(x) + \sum_{j=0}^N \hat{u}_j \sin((j+1)\pi x),$$

and use  $\tilde{u}_N \in V_N$  with homogeneous boundary values.  $B(x)$  can be

$$B(x) = u(1)x + u(0)(1-x)$$

such that

$$u_N(0) = B(0) = u(0) \quad \text{and} \quad u_N(1) = B(1) = u(1)$$

The variational problem becomes: Find  $\tilde{u}_N \in V_N$  such that

$$(B + \tilde{u}_N - u, v) = 0 \quad \forall v \in V_N$$

# Sines with boundary values

Find  $\tilde{u}_N \in V_N$  such that

$$(B + \tilde{u}_N - u, v) = 0 \quad \forall v \in V_N$$

Insert for  $\tilde{u}_N = \sum_{j=0}^N \hat{u}_j \sin((j+1)\pi x)$  and  $v = \sin((i+1)\pi x)$

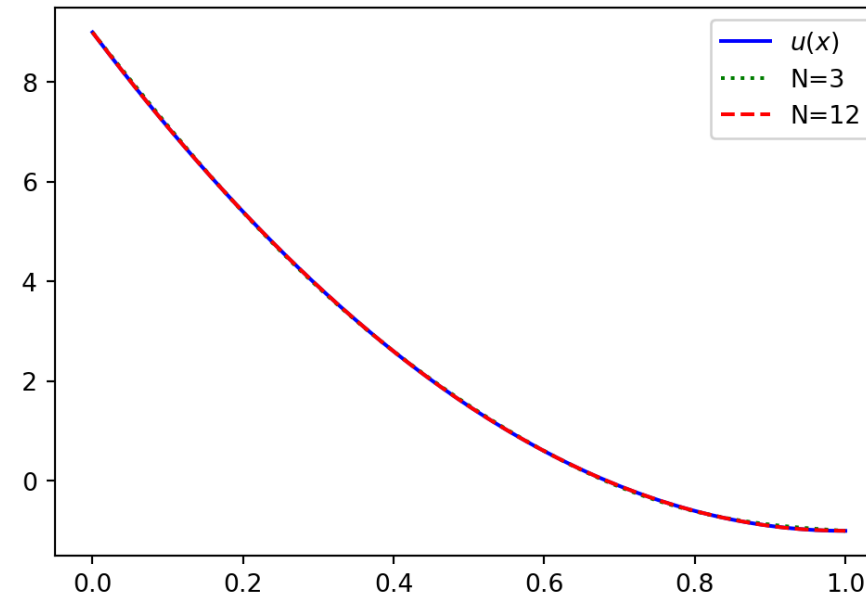
$$\left( \sum_{j=0}^N \hat{u}_j \sin((j+1)\pi x) + B - u, \sin((i+1)\pi x) \right) = 0, \quad i = 0, 1, \dots, N$$

and using the orthogonality of the sines we get

$$\hat{u}_i = 2(u - B, \sin((i+1)\pi x)) \quad i = 0, 1, \dots, N$$

# Result using boundary function $B(x)$

```
1 uc = []
2 B = u.subs(x, 0)*(1-x) + u.subs(x, 1)*x
3 for i in range(15):
4     uc.append(uhat(u-B, i).n())
5
6 uc = np.array(uc, dtype=float)
7 def uNc(uh, xj):
8     N = len(xj)
9     uj = u.subs(x, 0)*(1-xj) + u.subs(x, 1)*xj
10    for i, ui in enumerate(uh):
11        uj[:] += ui*np.sin((i+1)*np.pi*xj)
12    return uj
13
14 plt.figure(figsize=(6,4))
15 plt.plot(xj, 10*(xj-1)**2-1, 'b')
16 plt.plot(xj, uNc(uc[:3+1]), xj), 'g:')
17 plt.plot(xj, uNc(uc[:12+1]), xj), 'r--')
18 plt.legend(['$u(x)$', 'N=3', 'N=12']);
```



# How about the Legendre method?

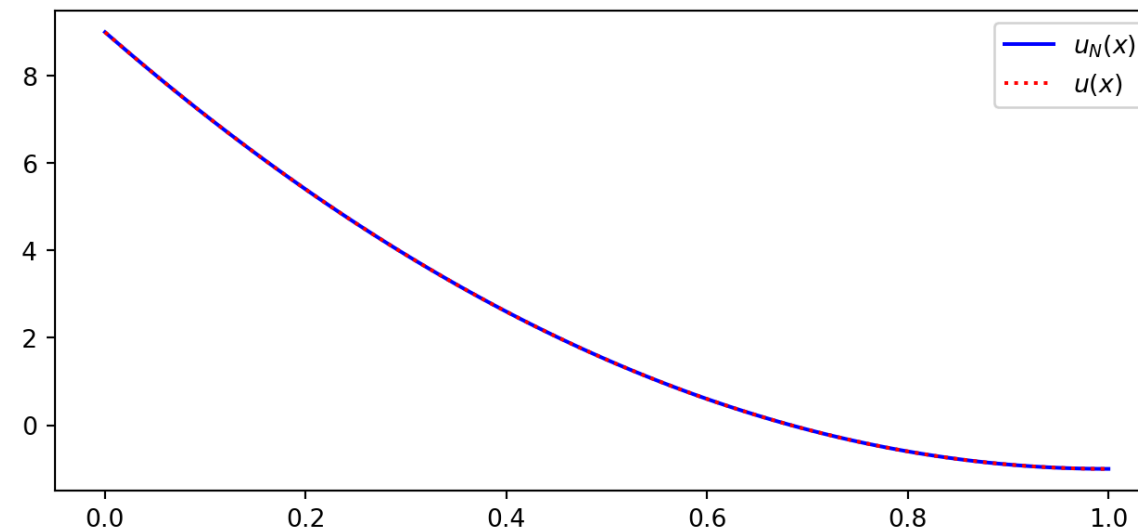
Legendre polynomials  $P_j(x)$  are such that the boundary values are nonzero

$$P_j(-1) = (-1)^j \quad \text{and} \quad P_j(1) = 1$$

Hence there is no need for boundary functions or adjustments

```
1 uhat = lambda u, j: (2*j+1) * inner(u, sp.legendre(j, x), (0, 1))/2
2 uL = [uhat(u, i) for i in range(6)]
3 plt.figure(figsize=(8, 3.5))
4 plt.plot(xj, Legendre(uL, domain=(0, 1))(xj), 'b')
5 plt.plot(xj, 10*(xj-1)**2-1, 'r:'); plt.legend([r'$u_N(x)$', r'$u(x)$'])
6 print(uL)
```

[7/3, -5, 5/3, 0, 0, 0]





# Summary

- We have been approximating  $u(x)$  by  $u_N \in V_N = \text{span}\{\psi_j\}_{j=0}^N$ , meaning that we have found  $u_N(x) = \sum_{j=0}^N \hat{u}_j \psi_j(x)$
- We have found the unknown expansion coefficients  $\{\hat{u}_j\}_{j=0}^N$  using
  - Variational methods
    - The least squares method
    - The Galerkin method
  - Collocation - an interpolation method
- Interpolation on uniform grids is bad for large  $N$  due to Runge's phenomenon
- Basis functions are often defined on a reference domain, and problems thus need mapping.
- Some basis functions, like sines, have zero boundary values and need an additional boundary function for convergence.
- Legendre polynomials lead to spectral convergence (for smooth functions  $u(x)$ ).