00_foundations

October 16, 2019

1 Applied Mathematics: an Introduction to Scientific Computing

Prof. Luca HELTAI

Prof. Gianluigi ROZZA

1.1 Basic Principles

Numerical Analysis is the *"art of approximating"*. Quoting Wikipedia:

> An approximation is an inexact representation of something that is still close enough to be useful. Although approximation is most often applied to numbers, it is also frequently applied to such things as mathematical functions, shapes, and physical laws.

Approximations should be used when incomplete information prevents use of exact representations. Many problems in physics are either too complex to solve analytically, or impossible to solve using the available analytical tools. Thus, even when the exact representation is known, an approximation may yield a sufficiently accurate solution while reducing the complexity of the problem significantly.

The type of approximation used depends on the available information, the degree of accuracy required, the sensitivity of the problem to this data, and the savings (usually in time and effort) that can be achieved by approximation.

In this course we focus on three main aspects:

- Methodologies (or approximation algorithms)
- Analysis (estimate errors and convergence properties)
- Implementation (through python and numpy notebooks)

Approximation is a matter of

- Representation (floating point values VS real numbers, finite dimensional spaces VS infinite dimensional ones, etc.)
- Measure of the Error (how do we know that we did a good job in approximating?)

In general, we will end up working with R^n. We recall here some basi principles:

1.1.1 Norms of vectors, matrices and functions

Given a vector space V over the field of real $\mathbb{R}$ or complex numbers $\mathbb{C}$ (V might be infinite dimensional), a *semi-norm* on V is a function $|\cdot| : V \to \mathbb{C}$ satisfying:

1. $|cf| = |c||f|$, for all $c \in \mathbb{C}$;
2. $|f + g| \leq |f| + |g|$ (often known as triangle inequality).

As it can be easily seen (1)–(2) imply that the norm is always non-negative:

$$0 = 0 \cdot |f| = |0 \cdot f| = |(1-1)f| = |f - f| \leq |f| + |(-1)f| = 2\ |f|.$$

The semi-norm becomes a *norm* if in addition to (1)–(2) we have also that for all $f \in V$

1. $|f| = 0$ if and only if $f = 0$,

A complete vector space with a norm is called a *Banach space*.

An inner product is any sesquilinear function $(\cdot, \cdot) : V \times V \mapsto \mathbb{C}$ satisfying the following conditions:

1. $(f, g) = \overline{(g, f)}$;
2. $(f, f) \geq 0$; $(f, f) = 0$ if and only if $f = 0$;
3. $(\alpha f, g) = \alpha(f, g)$ for all $\alpha \in \mathbb{C}$;
4. $(f + g, h) = (f, h) + (g, h)$.

The norm is then defined as $\|f\|^2 = (f, f)$. That this is a norm (i.e. satisfies the triangle inequality) is proved by first proving the *Cauchy-Schwarz* inequality

$$|(f, g)| \leq \sqrt{(f, f)(g, g)}.$$

The proof of the latter is as follows: For any $\alpha \in \mathbb{C}$ we have that

$$0 \leq (f - \alpha g, f - \alpha g) = (f, f) - \alpha(g, f) - \overline{\alpha}(f, g) + |\alpha|^2(g, g).$$

If $g = 0$ then the inequality is obviously true. If $g \neq 0$ we choose $\alpha = \frac{(f,g)}{(g,g)}$ to obtain the following

$$0 \leq (f, f) - \frac{|(f, g)|^2}{(g, g)}.$$

The triangle inequality then follows from the Schwarz inequality

$$(f + g, f + g) = \|f + g\|^2 = \|f\|^2 + \|g\|^2 + \mathrm{Re}[(f, g)] \leq (\|f\| + \|g\|)^2.$$

A Banach space with inner product and a norm induced by this product is called a *Hilbert space*.

Here are some examples of norms and semi-norms:

lp $V = \mathbb{R}^n$, and for any $x \in V$, its ℓ_p norm is defined as

$$\begin{aligned} \|x\|_p^p &= \sum_{i=1}^{n} |x_i|^p, \quad 1 \leq p < \infty, \\ \|x\|_\infty &= \sup_{1 \leq i \leq n} |x_i|. \end{aligned}$$

Replacing n by ∞ in the above equations leads to the definition of a Banach space denoted usually with ℓ_p, with elements finite or infinite sequences for which

$$\begin{aligned} \|x\|_p^p &= \sum_{i} |x_i|^p, \quad 1 \leq p < \infty, \\ \|x\|_\infty &= \sup_{i} |x_i|. \end{aligned}$$

are finite.

LP Let $I = (a, b)$. Then $L_p(I)$ is defined as the vector space of measurable functions f, for which

$$\begin{aligned} \|f\|_p &:= \left(\int_I |f|^p \, dx \right)^{1/p} < \infty, \quad 1 \le p < \infty, \\ \|f\|_\infty &:= \operatorname*{ess\,sup}_{x \in I} |f(x)| < \infty, \quad p = \infty. \end{aligned}$$

That the quantity defined in Example [ex:LP] is a norm one can show by using Hölder's and Minkowski's inequalities, which are as follows: Let $1 \le p \le \infty$, and q is the conjugate exponent to p (i.e. $p^{-1} + q^{-1} = 1$, with $q = \infty$, when $p = 1$). Then

$$\begin{aligned} \|fg\|_1 &\le \|f\|_p \|g\|_q, \quad \text{(Hölder's inequality)} \\ \|f + g\|_p &\le \|f\|_p + \|g\|_p, \quad \text{(Minkowski's inequality)}. \end{aligned}$$

Note that for $p = q = 2$ the Hölder's inequality is same as Schwarz inequality, by defining the inner product in L_2 as

$$(f, g) := \int_I f(x) g(x) \, dx.$$

Also note that the Minkowski's inequality is the triangle inequality in L_p. Similar inequalities hold true if the integrals are replaced by finite or infinite sums, and we also have Hölder's and Minkowski's inequalities for the spaces considered in example [ex:lp].

CK Let $I = [a, b]$ and $C^k(I)$ $k \in \mathbb{N}$ be the vector space of functions, whose derivatives of order $\le k$ are continuous. A (semi)-norm in $C^k(I)$ are then defined as

$$|f|_{k,\infty} = \|f^{(k)}\|_\infty = \sup_{x \in I} |f^{(k)}(x)|, \quad \|f\|_{W^k_\infty(I)} = \sup_{0 \le i \le k} |f|_{i,\infty,I}.$$

WKP For $1 \le p < \infty$, and $k \in \mathbb{N}$ we define the *Sobolev (semi)-norm*, as follows:

$$|f|_{k,p,I} := \|f^{(k)}\|_{0,p,I} = \|f^{(k)}\|_p, \quad \|f\|_{k,p,I} := \left(\sum_{0 \le i \le k} |f|^p_{i,p,I} \right)^{1/p}$$

The functions with finite Sobolev norm form the Banach space $W^k_p(I)$.

The examples above introduce several classes of Banach spaces, $C^k(I)$, $L_p(I)$ and W^k_p. They have a strightforward generalizations to higher dimensions.

For the normed finite dimensional spaces, the following result holds.

Let V be a finite dimensional vector space. Then all norms in V are equivalent.

and

Every finite dimensional space is closed.

1.1.2 Stability

In an abstract setting, we describe a generic problem as

$$F(x, d) = 0$$

where x is the unknown (generally a real number, a vector or a function) and $d \in D$ is the data. For each of the elements above we use an appropriate norm (see Lecture norms for a short introduction on norms and vector spaces), which will enable us to measure quantities of interests from a numerical point of view, such as errors, stability, and dependency of the solution from the data. In particular we will use the symbols $\|\cdot\|_F$, $\|\cdot\|_x$ and $\|\cdot\|_d$ to indicate the various norms.

In general, not all problems can be approximated. If we write a problem as above, then its approximation is useful only if the continuous problem has a unique solution which depends continuously on the data. We call these problems *well posed* or *stable*:

[def:cont-stability] A mathematical problem is *well posed* or *stable* if the following properties are satisfied:

- Uniqueness of solutions:
$$\forall d \in D, \exists! \, x \text{ s.t. } F(x, d) = 0.$$
- Continuous dependence on data:

Let δd be a perturbation of the data, such that $d + \delta d \in D$, and let $x + \delta x$ be the corresponding perturbed solution, i.e., $F(x + \delta x, d + \delta d) = 0$, then

$$\begin{aligned} &\forall d \in D, \quad \exists \eta_0(d), K_0 \text{ s.t.} \\ &\|\delta d\|_d < \eta_0 \in D \quad \Longrightarrow \quad \|\delta x\|_x < K_0 \|\delta d\|_d. \end{aligned}$$

1.1.3 Explanation for Dummies

The problem is considered stable is

Stable

$$\frac{||f(x - S_x) - f(x)||}{||S_x||} <= k$$

- *Where k is fixed and not depends on data*

ill-condition - problem is stable but k is **large**

Relative Stable

$$\frac{\frac{||f(x-S_x)-f(x)||}{||f(x)||}}{\frac{||S_x||}{||x||}} <= k$$

1.1.4 Condition numbers

A measure of how accurately we can approximate the problem at hand, is then given by the *Condition Number*:

Relative condition number:

$$K := \sup_{\delta d \text{ s.t. } d+\delta d \in D} \frac{\|\delta x\|_x / \|x\|_x}{\|\delta d\|_d / \|d\|_d}.$$

Absolute condition number (to be used when either $\|x\|_x = 0$ or $\|d\|_d = 0$):

$$K_{abs} := \sup_{\delta d \text{ s.t. } d+\delta d \in D} \frac{\|\delta x\|_x}{\|\delta d\|_d}.$$

If there exist a unique solution x to each data d, then we can construct a *resolvent map* G such that $G(d) = x$ and $F(G(d), d) = 0$. Assuming that G is differentiable, then a Taylor expansion of G around d allows us to express the condition numbers as

$$K \simeq \|G'(d)\| \frac{\|d\|_d}{\|G(d)\|_x}.$$

and

$$K_{abs} \simeq \|G'(d)\|.$$

A *stable* problem is *well conditioned* when its condition number is "small", where the meaning of "small" depends on the problem at hand.

1.1.5 Numerical stability

Once we have a *stable* problem, its approximation is usually given by a sequence of approximating problems

$$F_n(x_n, d_n) = 0, \qquad n \geq 1$$

such that

$$\begin{aligned} \lim_{n\to\infty} \|F_n - F\|_F &= 0 \\ \lim_{n\to\infty} \|x_n - x\|_x &= 0 \\ \lim_{n\to\infty} \|d_n - d\|_d &= 0, \end{aligned}$$

for some appropriate norms.

Equivalently to what happens in the continuous case, we can establish the stability of the approximate n-th problem.

A mathematical approximation of a stable problem is itself *stable* if the following properties are satisfied:

- Uniqueness of solutions:

$$\forall n \geq 1, \forall d_n \in D_n, \exists! \, x_n \text{ s.t. } F_n(x_n, d_n) = 0.$$

- Continuous dependence on data:

Let δd_n be a perturbation of the data, such that $d_n + \delta d_n \in D_n$, and let $x_n + \delta x_n$ be the corresponding perturbed solution, i.e., $F_n(x_n + \delta x_n, d_n + \delta d_n) = 0$, then

$$\begin{aligned} &\forall d_n \in D_n, \quad \exists \eta_n(d), K_n \text{ s.t.} \\ &\|\delta d_n\|_d < \eta_n \in D_n \quad \Longrightarrow \quad \|\delta x_n\|_x < K_n \|\delta d_n\|_d. \end{aligned}$$

1.1.6 Consistency

Whenever the data d is admissible for F_n, then further properties of the approximations can be devised. In particular,

A numerical problem is *consistent*, when, assuming $d \in D_n \quad \forall n$,

$$\lim_{n \to \infty} F_n(x, d) = \lim_{n \to \infty} F_n(x, d) - F(x, d) = 0.$$

Moreover,

A numerical approximation is *strongly consistent* when

$$F_n(x, d) = 0, \qquad \forall n.$$

1.1.7 Convergence

A numerical method is *convergent* when

$$\begin{aligned} &\forall \varepsilon > 0, \qquad \exists \, n_0(\varepsilon), \exists \, \delta(\varepsilon, n_0) \text{ s.t.} \\ &\forall n > n_0, \quad \forall \delta d_n : \|\delta d_n\|_d < \delta \quad \Longrightarrow \quad \|x(d) - x_n(d + \delta d_n)\| < \varepsilon, \end{aligned}$$

where $x(d)$ is the solution to $F(x, d) = 0$ and $x_n(d + \delta d_n)$ is the solution to $F_n(x_n, d + \delta d_n)$.

A convergent approximation is always stable.

1.1.8 Lax-Richtmyer theorem

One of the fundamental theorem of numerical analysis is the so called Lax-Richtmyer theorem:

> If a problem is consistent, then stability and convergence are equivalent.

1.1.9 Examples of Stable Problems

The problem of finding the solution $x \in R^n$ to the linear system of equations $Ax = d$, where $d \in R^n$ and $A \in R^{n \times n}$ can be written in the form $F(x, d) = 0$ simply by defining $F(x, d) := Ax - d$.

These problems are well defined if and only if the matrix A is invertible. In these cases, the resolvant $G(x)$ is the multiplication with the inverse of the matrix itself, i.e., if $F(G(d), d) = 0$ then it must be $G(d) = A^{-1}d$, and in general we have:

$$A(x + \delta x) = d + \delta d$$

$$\begin{aligned} A\delta x &= \delta d \\ \|A\delta x\| &= \|\delta d\| \end{aligned}$$

$$\begin{aligned} \delta x &= A^{-1}\delta d \\ \|\delta x\| &= \|A^{-1}\delta d\| \end{aligned}$$

$$\begin{aligned} \|\delta x\| &\leq \|A^{-1}\| \quad \|\delta d\| \\ \|d\| &\leq \|A\| \quad \|x\| \end{aligned}$$

$$\|\delta x\|/\|x\| \leq \|A^{-1}\|\|A\| \quad \|\delta d\|/\|d\|$$

We see that the **absolute condition number** of this problem is then equal to $\|A^{-1}\|$, and the relative condition number is instead equal to $| A^{-1} || A |$. This is what is usually called the **condition number** of a matrix (if nothing is said, it is intended that we are talking about the relative condition number).

[1]:
```
import numpy as np
from numpy.linalg import solve, norm, cond

# Construct a random matrix, and check its condition number (fix random seed,␣
 ↪so we have always the same number)
np.random.seed(101)
A = np.random.rand(100,100)

K = cond(A) # Linear algebra package of numpy

print(K)
```

```
1933.14691697
```

The condition number expresses the worst case scenario in relative error we should expect when solving linear systems. If we perturb the data with a unit vector, we should expect the new solution to be at distance K from the original solution...

[2]:
```
# Construct an artificial solution
x = np.random.rand(100)

d = A.dot(x); # We could use A*x only if we created a Matrix! This is an ndarray

# Verify that we got the right thing...
x_test = solve(A,d)
error = norm(x_test-x)
print("Initial Error", error)

# Now perturb d with a unit perturbation, and check the norm of the new solution
delta_d = np.random.rand(100)
delta_d /= np.linalg.norm(delta_d)

xnew = np.linalg.solve(A,d+delta_d)
delta_x = xnew-x

deviation = np.linalg.norm(delta_x)
print("Absolute deviation: ", deviation)

relative_deviation = (norm(delta_x)/norm(x))/(norm(delta_d)/norm(d))
print("Relative deviation: ", relative_deviation)
```

```
Initial Error 2.37596167042e-13
Absolute deviation:  0.676705032761
Relative deviation:  29.8201041662
```

We see in this case that, upon a unit norm perturbation in the data, we obtained a relative perturbation of about 50. The upper limit of this perturbation is given by the relative condition number. The bigger the condition number, the more sensitive to perturbations in the data will be your solution!

01a_interpolation-2d

October 16, 2019

0.1 Two dimensional Lagrange interpolation

To extend to the two dimensional case, we start from two sets of distinct points, say $(n+1)$ points in the x direction, and $m+1$ points in the y direction, in the interval $[0,1]$.

A two dimensional version of the the Lagrange interpolation is then used to constuct polynomial approximations of functions of two dimensions. A polynomial from $\Omega := [0,1]\times[0,1]$ to R is defined as:

$$\mathcal{P}^{n,m} : \text{span}\{p_i(x)p_j(y)\}_{i,j=0}^{n,m}$$

and each *multi-index* (i,j) represents a polynomial of order $i+j$, i along x, and j along y. For convenience, we define

$$p_{i,j}(x,y) := p_i(x)p_j(y) \qquad i=0,\ldots,n \qquad j=0,\ldots,m$$

Alternatively we can construct a basis starting from the Lagrange polynomials:

$$l_{i,j}(x,y) := l_i(x)l_j(y) \qquad i=0,\ldots,n \qquad j=0,\ldots,m$$

where we use the same symbol for the polynomials along the two directions for notational convenience, even though they are constructed from two different sets of points.

We define the *Lagrange interpolation* operator $\mathcal{L}^{n,m}$ the operator

$$\mathcal{L}^{n,m} : C^0([0,1]\times[0,1]) \mapsto \mathcal{P}^{n,m}$$

which satisfies

$$(\mathcal{L}^{n,m}f)(q_{i,j}) = f(q_{i,j}), \qquad i=0,\ldots,n, \qquad q_{i,j} := (x_i, y_j)$$

In order to prevent indices bugs, we define two different refinement spaces, and two different orders, to make sure that no confusion is done along the x and y directions.

We try to be dimension independent, so we define everything starting from tuples of objects. The dimension of the tuple defines if we are working in 1, 2, or 3 dimensions.

[1]:
```
%matplotlib inline
from numpy import *
from pylab import *
```

```
dim = 2
ref = (301, 311)
n   = (4,5)

assert dim == len(ref) == len(n), 'Check your dimensions!'

x = [linspace(0,1,r) for r in ref]
q = [linspace(0,1,r+1) for r in n]
```

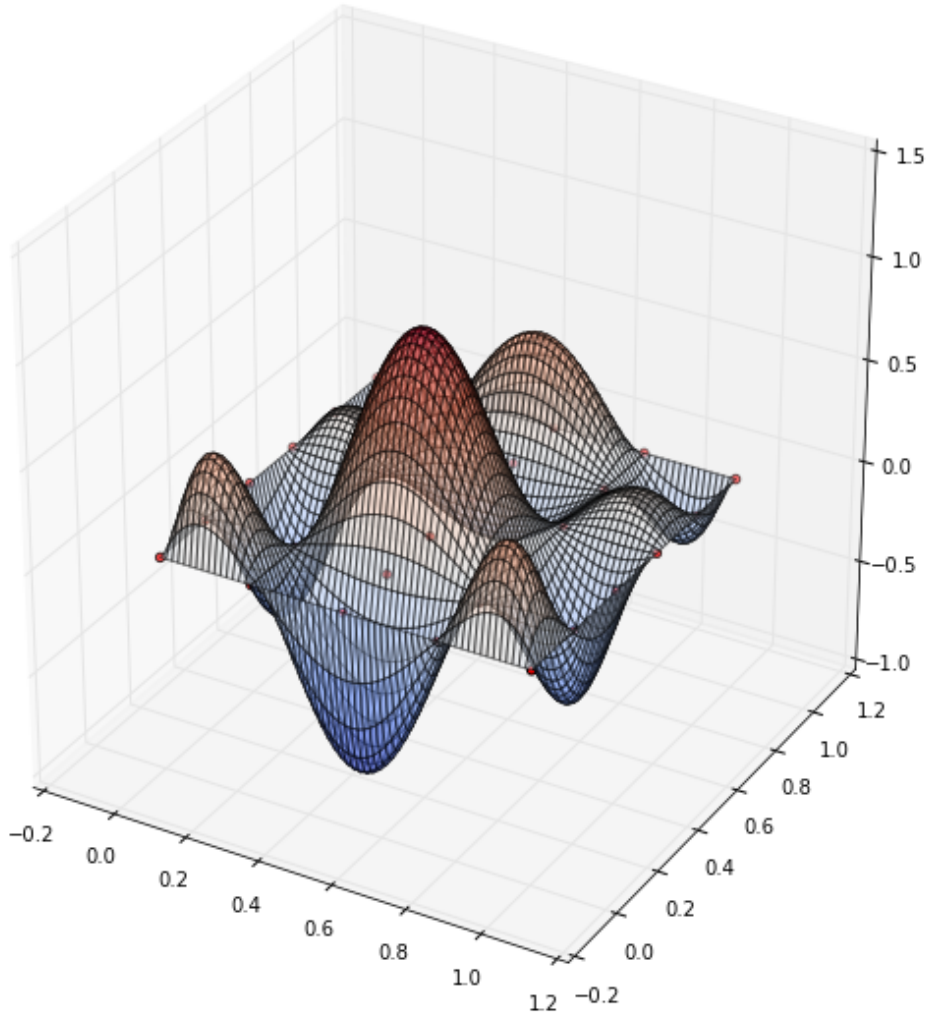
We start by constructing the one dimensional basis, for each dimension. Once this is done, we compute the product $l_i(x)l_j(y)$, for each x and y in the two dimensional list x, containing the x and y points. This product can be reshaped to obtain a matrix of the right dimension, provided that we did the right thing in broadcasting the dimensions...

[2]:
```
Ln = [zeros((n[i]+1, ref[i])) for i in xrange(dim)]

# Construct the lagrange basis in all directions
for d in xrange(dim):
    for i in xrange(n[d]+1):
        Ln[d][i] = product([ (x[d]-q[d][j])/(q[d][i]-q[d][j]) for j in␣
 ↪xrange(n[d]+1) if j != i], axis=0)

# Now construct the product between each basis in
# each coordinate direction, to use for plotting and interpolation
if dim == 2:
    L = einsum('ij,kl -> ikjl', Ln[1], Ln[0])
elif dim == 3:
    L = einsum('ij,kl,mn -> ikmjln', Ln[2], Ln[1], Ln[0])
elif dim == 1:
    L = Ln[0]
else:
    raise

print(L.shape)
Lf = reshape(L, (prod(L.shape[:dim]), prod(L.shape[dim:])))
print(Lf.shape)
```

```
(6, 5, 311, 301)
(30, 93611)
```

[3]:
```
from mpl_toolkits.mplot3d import Axes3D

X = meshgrid(x[0], x[1])
Q = meshgrid(q[0], q[1])

fig = figure(figsize=[10,10])
```

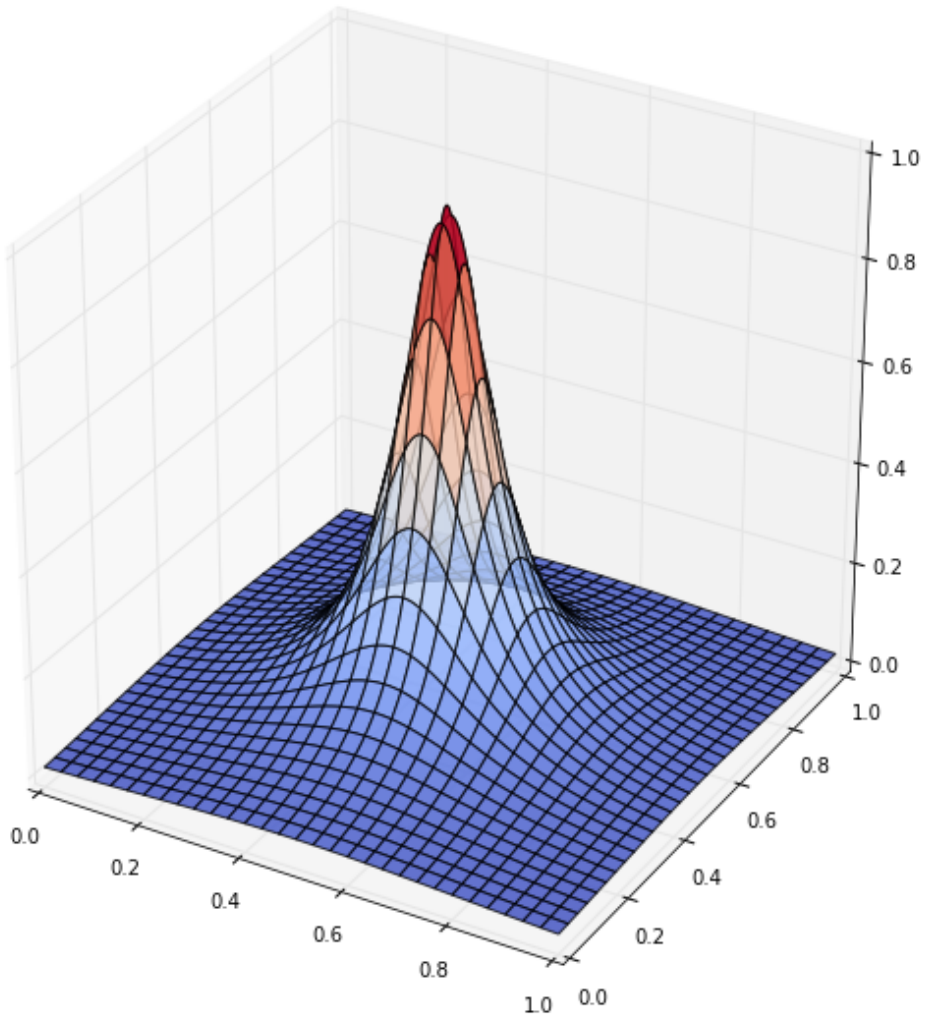
```
ax = fig.gca(projection='3d')
surf2 = ax.plot_surface(X[0], X[1], L[2,2], rstride=5, cstride=5, cmap=cm.
 ↪coolwarm, alpha=0.5)
scatter = ax.scatter(Q[0], Q[1], zeros_like(Q[0]), c='r', marker='o')
```

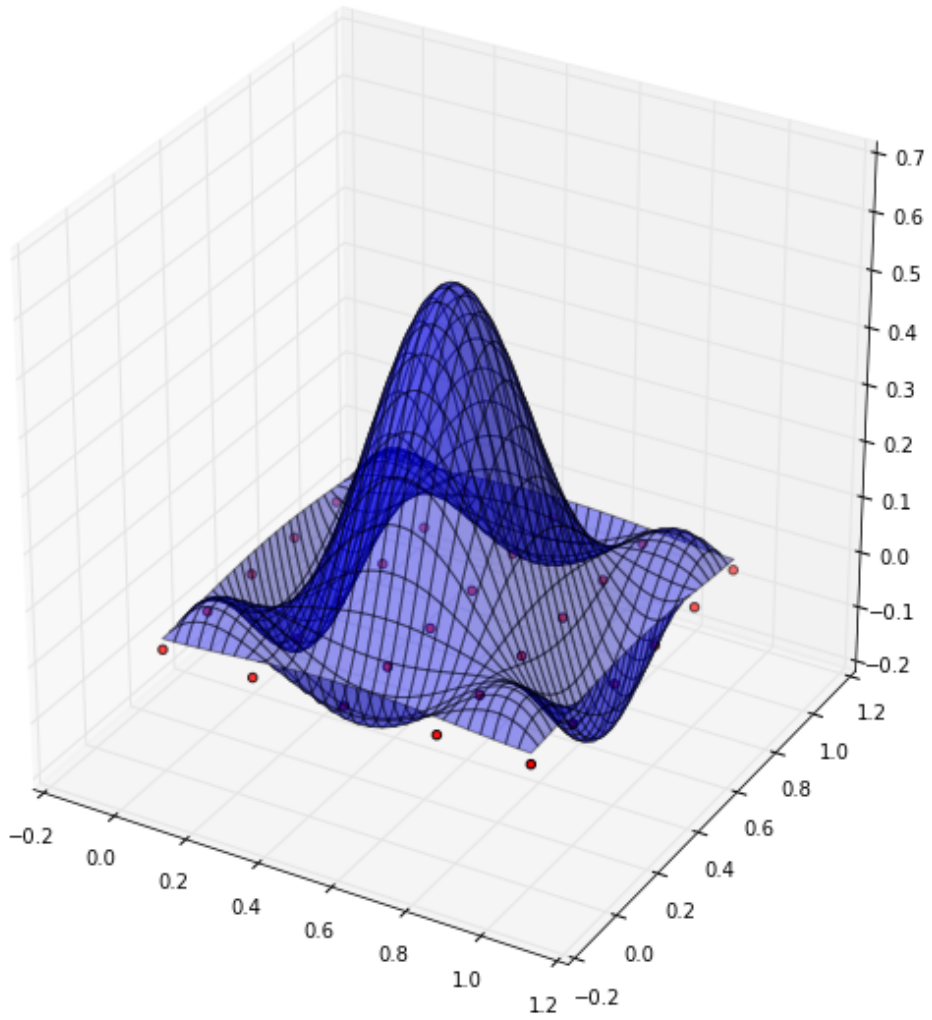


Now we try to make an interpolation. First we need to evaluate the function at the interpolation points. This is done by expressing all possible combinations of the points by meshgrid on q:

[4]:
```
Q = meshgrid(q[0], q[1])

def f(x,y):
    return 1/(1+100*((x-.5)**2+(y-.5)**2))
```

```
def my_plot(f):
    fig = figure(figsize=[10,10])
    ax = fig.gca(projection='3d')
    surf2 = ax.plot_surface(X[0], X[1], f(X[0], X[1]), cmap=cm.coolwarm,␣
 ↪alpha=0.8)

    show()

    fig = figure(figsize=[10,10])
    ax = fig.gca(projection='3d')
    scatter = ax.scatter(Q[0], Q[1], zeros_like(Q[0]), c='r', marker='o')

    F = f(Q[0], Q[1])
    interp = Lf.T.dot(F.reshape((-1,))).reshape(X[0].shape)
    surf3 = ax.plot_surface(X[0], X[1], interp, alpha=0.4)
    show()

my_plot(f)
```


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0


0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2

01b_interpolation_properties

October 16, 2019

1 Properties of the Interpolating Polynomial

We fix the interpolation nodes $q_i, i = 0, \ldots, n$, and an interval $[a, b]$ containing all the interpolation nodes. The process of interpolation maps the function f to a polynomial p in $\mathcal{P}^n$. This defines a mapping $\mathcal{L}^n$ from the space $C^0([a, b])$ of all continuous functions on $[a, b]$ to itself. The map $\mathcal{L}^n$ is linear and it is a **projection** on the subspace $\mathcal{P}^n$ of polynomials of degree n or less.

1.1 Interpolation error for smooth functions

If f is $n+1$ times continuously differentiable on a closed interval I and $p := \mathcal{L}^n f$ is the polynomial of degree at most n that interpolates f at the distinct points $\{q_i\}_{i=0}^n$ in that interval, then for each x in the interval there exists ξ in that interval such that

$$e(x) := f(x) - p(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} w(x) \qquad w(x) := \prod_{i=0}^{n} (x - q_i).$$

The above error bound suggests choosing the interpolation points such that the product $|\prod_{i=0}^n (x - q_i)|$, is as small as possible.

1.1.1 Proof

Fix $x \in [a, b]$, and consider the function

$$G(t) = (f(x) - p(x))w(t) - (f(t) - p(t))w(x)$$

The function $G(t)$ has exactly $n + 2$ zeros in the interval $[a, b]$ (the set of zeros of $w(t)$, plus the point x). By Rolle's theorem, there exists $\xi \in (a, b)$ such that

$$\frac{d^{(n+1)} G(\xi)}{dt^{(n+1)}} = 0$$

The function $G^{(n+1)}$ is given by

$$G^{(n+1)}(t) = (f(x) - p(x))(n + 1)! - f(t)^{(n+1)} w(x)$$

and the thesis follow, by computing it in ξ.

1.2 Error estimates in the L^∞ norm

Given the theorem above, we then conclude that

$$\|f - \mathcal{L}^n f\|_\infty \leq \|f^{(n+1)}\|_\infty \left\| \frac{w}{(n+1)!} \right\|_\infty$$

The estimate above can be improved if the function f is *analytically extendible* in a oval $O(a, b, R)$ with $R > 0$.

In this case, it is possible to show that the $(n+1)$ derivative of f is controlled by f itself as

$$\|f^{(n+1)}\|_{\infty,\overline{O(a,b,R)}} \leq \frac{(n+1)!}{R^{n+1}} \|f\|_{\infty,\overline{O(a,b,R)}}$$

and therefore we have:

Theorem

$$\|f - \mathcal{L}^n f\|_\infty \leq \|f\|_{\infty,\overline{O(a,b,R)}} \left(\frac{|b-a| = 1}{R} \right)^{n+1}$$

The counter example of Runge:

$$f(x) = \frac{1}{1+x^2}$$

is an example of function which is $C^\infty(R)$, but it is not analytically extendible in the entire complex plane. In fact, the radius of the oval where f is analytically extendible is 1 (the numerator goes to zero when $z = \pm i$).

In this case, if the interpolation interval is larger than $R = 1$, then only the first result is valid.

1.3 Relation to best approximation

If p is the best possible approximation of f in $\mathcal{P}^n$, i.e.,

$$E(f) = \|f - p\|_\infty \leq \inf_{q \in \mathcal{P}^n} \|f - q\|_\infty,$$

then the relation with the Lagrangian interpolation is given by

$$\begin{aligned} \|f - \mathcal{L}^n f\|_\infty &= \|f - p + p - \mathcal{L}^n f\|_\infty \\ &= \|f - p + \mathcal{L}^n (f - p)\|_\infty \\ &\leq (1 + \|\mathcal{L}^n\|) \|f - p\|_\infty \end{aligned}$$

where the norm $\|\mathcal{L}^n\|$ is defined as

$$\|\mathcal{L}^n\| := \sup_{f\in C^0([0,1]),\quad \|f\|_\infty\leq 1} \|\mathcal{L}^n f\|_\infty \leq \|\sum_{i=0}^{n} |l_i|\|_\infty := \|\Lambda(\{q\}_{i=0}^n)\|_\infty$$

and

$$\Lambda(q) := \sum_{i=0}^{n} |l_i|$$

is often called the **Lebesgue function**.

In other words, the interpolation polynomial is at most a factor ($\|\mathcal{L}^n\| + 1$) worse than the best possible approximation. This suggests that we should look for a set of interpolation nodes that makes $\|\mathcal{L}^n\|$ small.

The **Chebyshev nodes** are the roots of the Chebyshev polynomials of the first kind. They are explicitly defined as

$$x_k = \cos\left(\frac{2k-1}{2n}\pi\right) \ , \ k = 1, \ldots, n$$

and they minimize the operatorial norm $\|\mathcal{L}^n\|$ on the interval $[-1, 1]$. In particular, we have for Chebyshev nodes on $[-1, 1]$:

$$\|\mathcal{L}^n\| \ \leq \frac{2}{\pi}\log(n+1) + 1.$$

Compare this to equidistant nodes:

$$\|\mathcal{L}^n\| \ \leq \frac{2^{n+1}}{cn\log(n)}$$

We conclude that Chebyshev nodes are a good choice for polynomial interpolation, as the growth in n is exponential for equidistant nodes. Notice however that $\|\mathcal{L}^n\|$ is still diverging when $n \to \infty$.

1.4 Convergence properties

It is natural to ask, for which classes of functions and for which interpolation nodes the sequence of interpolating polynomials converges to the interpolated function as $n \to \infty$? Convergence may be understood in different ways, e.g. pointwise, uniform or in some integral norm.

The situation is rather bad for equidistant nodes, in that uniform convergence is not even guaranteed for infinitely differentiable functions. One classical example, due to Carl Runge, is the function $f(x) = \frac{1}{1+x^2}$ on the interval $[-5, 5]$. The interpolation error $\|f - \mathcal{L}^n f\|_\infty$ grows without bound as $n \to \infty$. Another example is the function $f(x) = |x|$ on the interval $[-1, 1]$, for which the interpolating polynomials do not even converge pointwise except at the three points $x = +1, -1, 0$.

One might think that better convergence properties may be obtained by choosing different interpolation nodes. The following result seems to give a rather encouraging answer:

Theorem. For any function $f(x)$ continuous on an interval $[a, b]$ there exists a table of nodes for which the sequence of interpolating polynomials $p_n(x) = \mathcal{L}^n f$ converges to $f(x)$ uniformly on $[a, b]$.

Proof. It's clear that the sequence of polynomials of best approximation $p_n(x)$ converges to $f(x)$ uniformly (due to Weierstrass approximation theorem, shown later on). Now we have only to show that each $p_n(x)$ may be obtained by means of interpolation on certain nodes. But this is true due to a special property of polynomials of best approximation known from the **Chebyshev alternation theorem**. Specifically, we know that such polynomials should intersect $f(x)$ at least $n + 1$ times. Choosing the points of intersection as interpolation nodes we obtain the interpolating polynomial coinciding with the best approximation polynomial.

The defect of this method, however, is that interpolation nodes should be calculated anew for each new function $f(x)$, but the algorithm is hard to be implemented numerically. Does there exist a single table of nodes for which the sequence of interpolating polynomials converge to any continuous function $f(x)$? The answer is unfortunately negative:

Theorem. For any table of nodes there is a continuous function $f(x)$ on an interval $[a, b]$ for which the sequence of interpolating polynomials diverges on $[a, b]$.

The proof essentially uses the lower bound estimation of the Lebesgue constant, which we defined above to be the operator norm of $\mathcal{L}^n$. Now we seek a table of nodes for which

$$\lim_{n\to\infty} \mathcal{L}^n f = f, \qquad \forall f \in C^0([a, b]).$$

Due to the Banach–Steinhaus theorem, this is only possible when norms of $\mathcal{L}^n$ are uniformly bounded, which cannot be true since we know that

$$\mathcal{L}^n \geq \frac{2}{\pi} \log(n + 1) + C.$$

For example, if equidistant points are chosen as interpolation nodes, the function from Runge's phenomenon demonstrates divergence of such interpolation. Note that this function is not only continuous but even infinitely times differentiable on R. For the Chebyshev nodes, however, such an example is much harder to find due to the following result:

Theorem. For every absolutely continuous function on $[a, b]$, the sequence of interpolating polynomials constructed on Chebyshev nodes converges to $f(x)$ uniformly.

1.5 Weierstrass approximation theorem

Let B_n be a sequence of linear positive operators such that B_n converges uniformly to f, $\forall f \in \mathcal{P}^2$.

Then $B_n f$ converges uniformly to f for all $f \in C^0([a, b])$.

Proof

The idea of the proof is that for any $f \in C^0([a, b])$, for any $\varepsilon > 0$, and for each $x_0 \in [a, b]$, we can find a quadratic function $q > f$ s.t. $|q(x_0) - f(x_0)| < \varepsilon$.

Then there exists a $\bar{n}$ such that for $n > \bar{n}$, $|(B_n q)(x_0) - q(x_0)| < \varepsilon$, that is, $|(B_n q)(x_0) - f(x_0)| < 2\varepsilon$. If we can show that this is valid uniformly with respect to x_0, we have done.

Since f is continuous on $[a,b]$ compact, f is uniformly continuous:

$$\forall \varepsilon > 0, \quad \exists \delta \text{ s.t. } |f(x_1) - f(x_2)| < \varepsilon \text{ if } |x_1 - x_2| \leq \delta.$$

Then, chosen x_0, set

$$q^{\pm}(x) := f(x_0) \pm \left(\varepsilon + \frac{2\|f\|_\infty}{\delta^2}(x - x_0)^2 \right)$$

Such $q^{\pm}$ can be expressed as $ax^2 + bx + c$ where $|a|, |b|, |c| \leq M$ with M depending on $\|f\|_\infty, \varepsilon, \delta$, but **not** on x_0.

We have, by construction that $q^-(x) \leq f(x) \leq q^+(x)$ for all $x \in [a,b]$.

Choose N large enough so that

$$\|B_n x^i - x^i\|_\infty \leq \frac{\varepsilon}{M} \qquad i = 0, 1, 2$$

In this way we have

$$\|B_n q^{\pm} - q^{\pm}\|_\infty \leq 3\varepsilon$$

At this point it is easy to show that

$$\begin{aligned} (B_n f)(x_0) &\leq (B_n q^+)(x_0) \leq q^+(x_0) + 3\varepsilon = f(x_0) + 4\varepsilon \\ (B_n f)(x_0) &\geq (B_n q^-)(x_0) \leq q^-(x_0) - 3\varepsilon = f(x_0) - 4\varepsilon \end{aligned}$$

that is

$$-4\varepsilon \leq (B_n f)(x_0) - f(x_0) \leq 4\varepsilon \qquad \forall x_0 \qquad \Rightarrow \qquad \|B_n f - f\|_\infty \leq 4\varepsilon$$

01c_interpolation_error

October 16, 2019

1 Task for today:

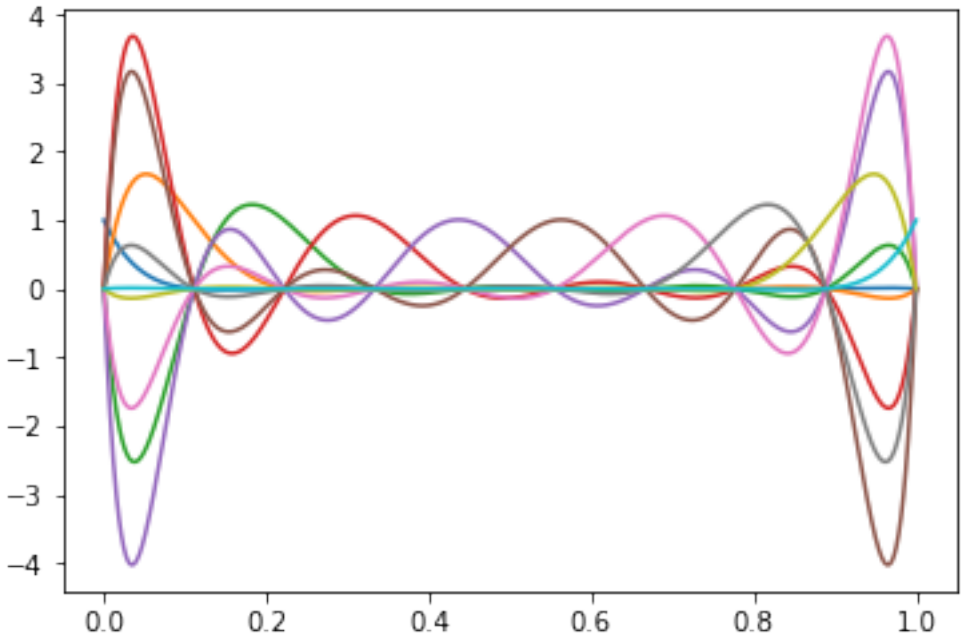
- compute the error between Lagrange interpolation for equispaced points (in "approximate Linfty") and a given function when the degree increases
- compute the error between Lagrange interpolation for Chebyshev (in "approximate Linfty") and a given function when the degree increases
- compute the error between Bernstein approximation (in "approximate Linfty") and a given function when the degree increases
- compute the L2 projection and compute the error ("in approximate Linfty") norm and compare with previous results

[1]:
```
%pylab inline
```

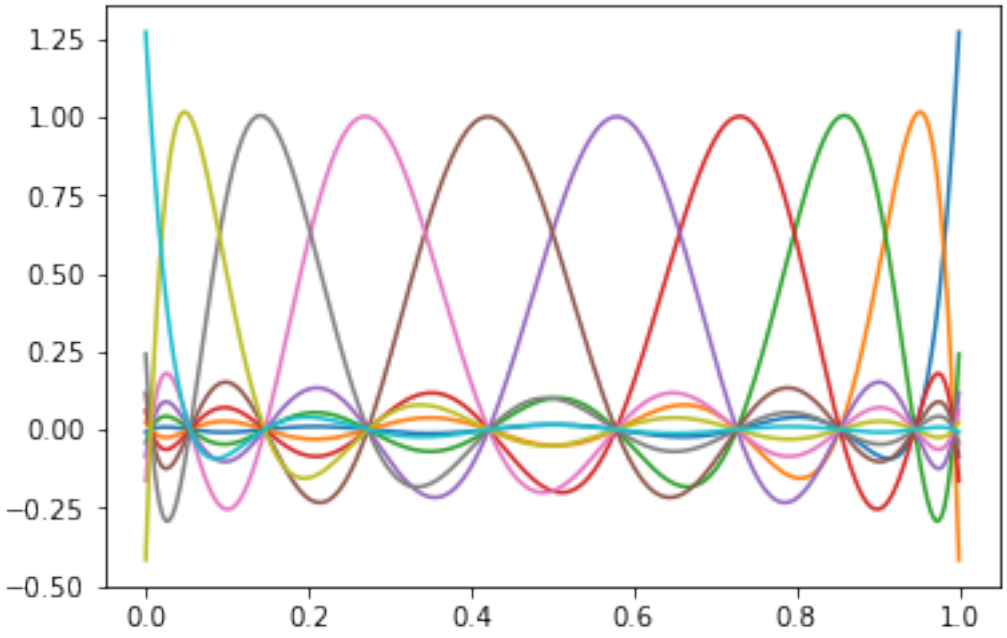
```
Populating the interactive namespace from numpy and matplotlib
```

[2]:
```
import scipy.special

def lagrange(i, q, x):
    return product([(x-qj)/(q[i]-qj) for qj in q if qj != q[i]], axis=0)

def bernstein(i, q, x):
    n = len(q)-1
    return scipy.special.binom(n,i)*(1-x)**(n-i)*x**i

def cheb(n):
    return numpy.polynomial.chebyshev.chebgauss(n)[0]*.5+.5
```

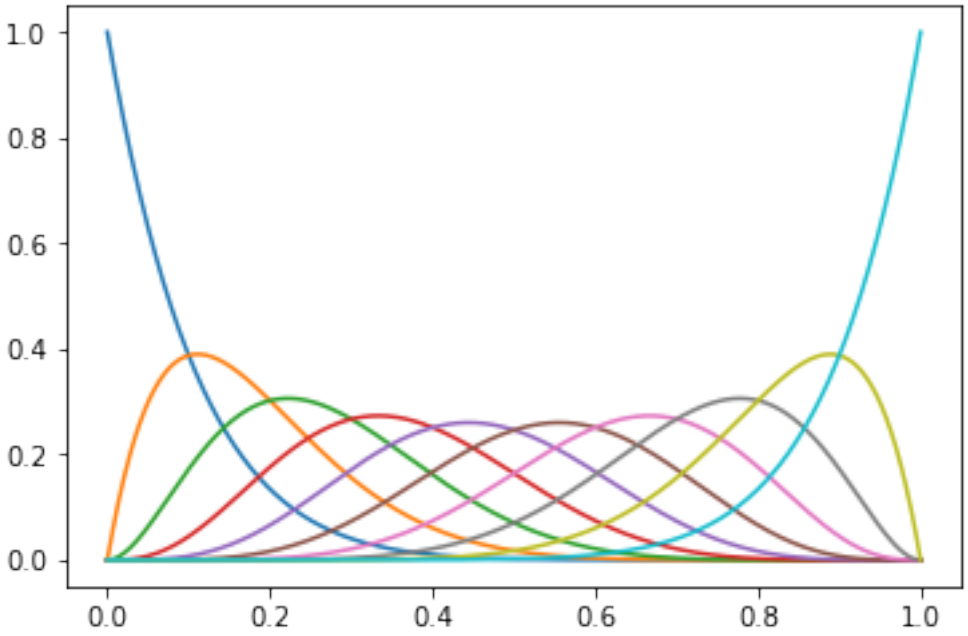
[3]:
```
x = linspace(0,1,1025)
q = linspace(0,1,10)
y = array([lagrange(i,q,x) for i in range(len(q))])
_ = plot(x, y.T)
```



[4]:
```
q = cheb(10)
y = array([lagrange(i,q,x) for i in range(len(q))])
_ = plot(x, y.T)
```

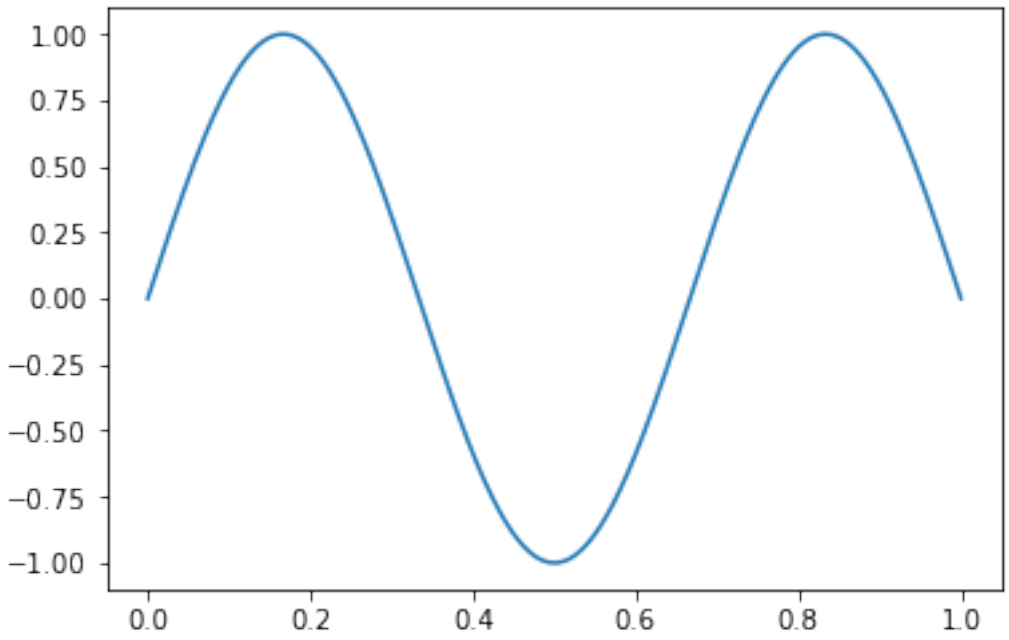


[5]:
```
q = linspace(0,1,10)
y = array([bernstein(i,q,x) for i in range(len(q))])
_ = plot(x, y.T)
```



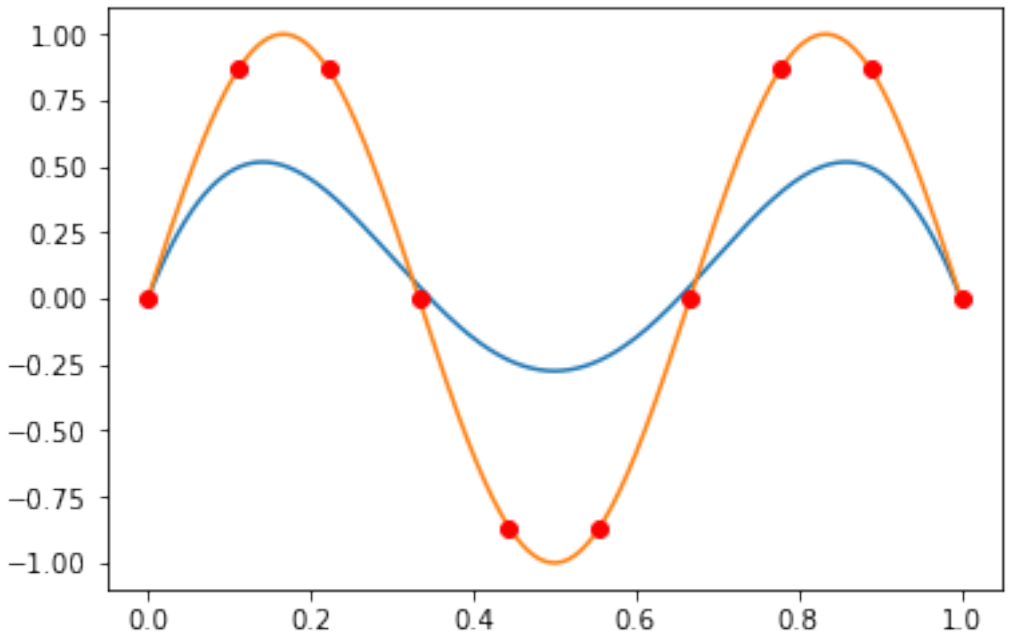
[6]:
```
def myfun(x):
    return 1/(1+100*(x-.5)**2)

def myfun(x):
    return sin(3*numpy.pi*x)


plot(x, myfun(x))
```

[6]: [<matplotlib.lines.Line2D at 0x120e37a20>]



[7]:
```
p = y.T.dot(myfun(q))
f = myfun(x)

plot(x,p)
plot(x,f)
plot(q,myfun(q), 'or')
```

[7]: [<matplotlib.lines.Line2D at 0x120f318d0>]



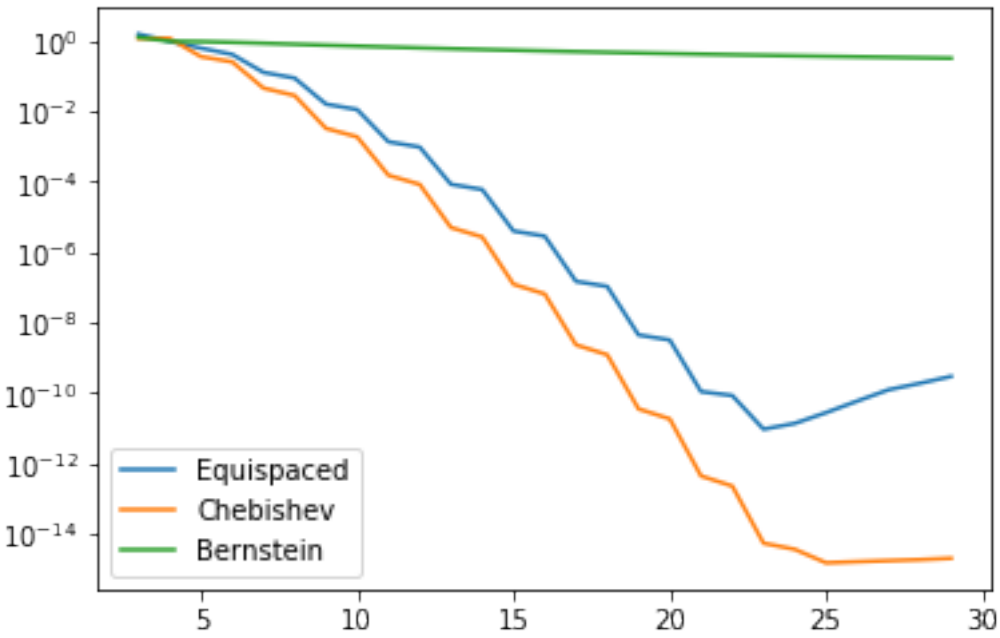
[8]:
```
linfty = max(abs(f-p))
linfty
```

[8]: 0.72598414958382995

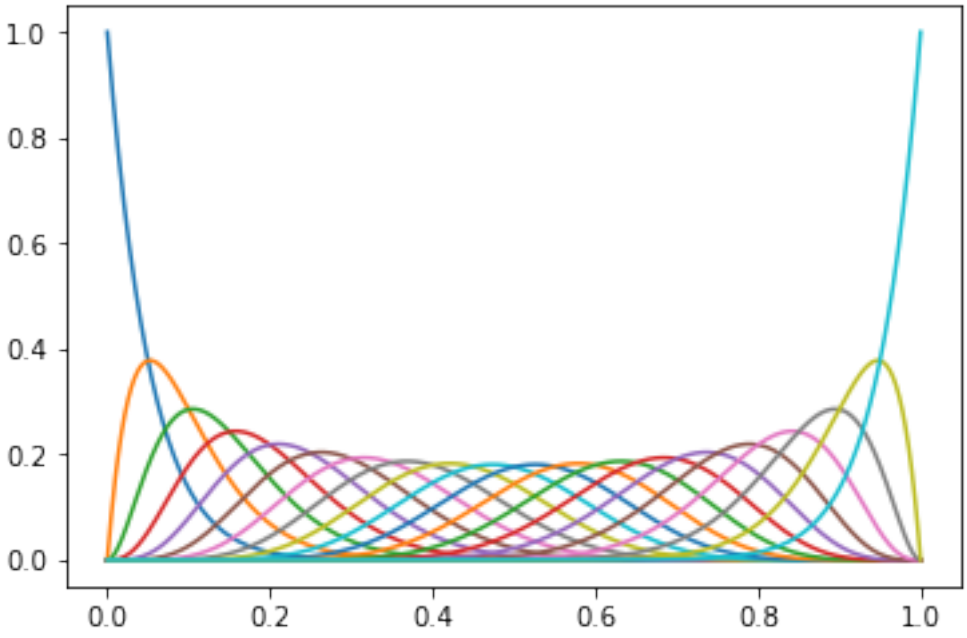
[9]:
```
def error(q, myfun, interpolation=lagrange) :
    y = array([interpolation(i,q,x) for i in range(len(q))])
    p = y.T.dot(myfun(q))
    f = myfun(x)
    return (max(abs(f-p)))
```

[10]:
```
N = range(3, 30)
error_equispaced = []
error_cheb = []
error_bernstein = []
for n in N:
    error_cheb.append(error(cheb(n), myfun))
    error_equispaced.append(error(linspace(0,1,n), myfun))
    error_bernstein.append(error(linspace(0,1,n), myfun, bernstein))
```

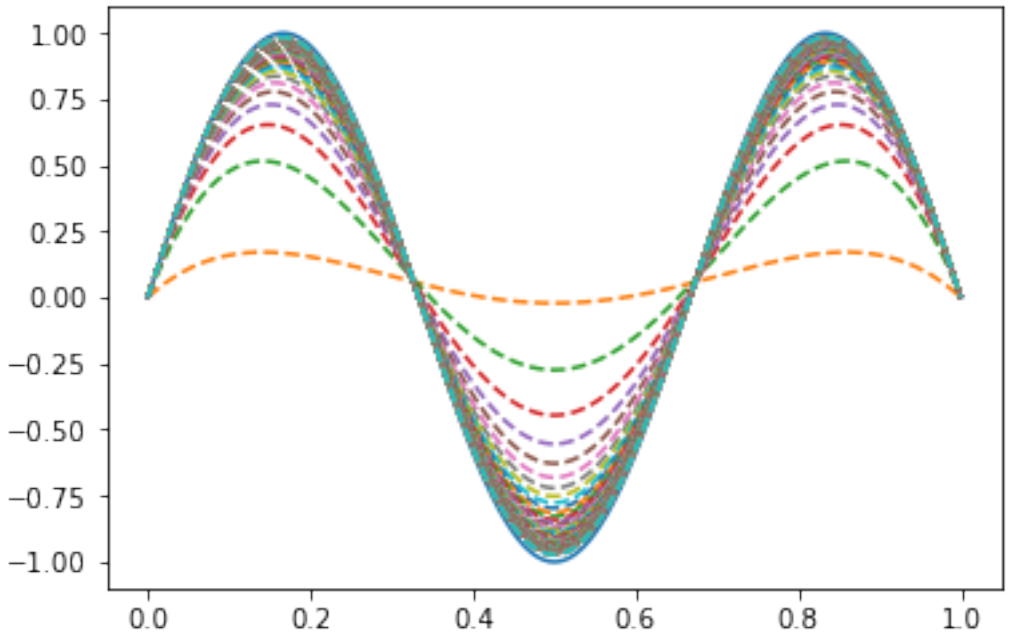
[11]:
```
semilogy(N, error_equispaced)
semilogy(N, error_cheb)
semilogy(N, error_bernstein)
_ = legend(['Equispaced','Chebishev','Bernstein'])
```



[12]:
```
q = linspace(0,1,20)
y = array([bernstein(i,q,x) for i in range(len(q))])
_ = plot(x, y.T)
```



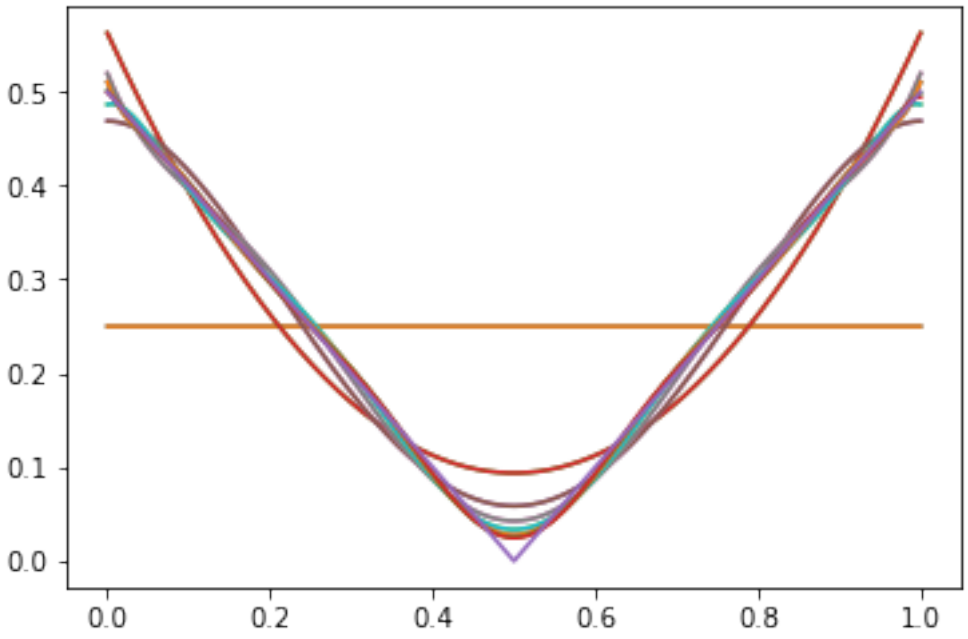
[13]:
```
N = range(5,400,5)
plot(x,myfun(x))

for n in N:
    q = linspace(0,1,n)
    y = array([bernstein(i,q,x) for i in range(len(q))])
    p = y.T.dot(myfun(q))
    _ = plot(x, p, '--')
```



[14]:
```
def myfun(x):
    return abs(x-.5)

import scipy
from scipy.integrate import quad as integrate

N = range(1,15)

for n in N:
    M = zeros((n,n))

    for i in range(n):
        for j in range(n):
            M[i,j] = 1.0/(i+j+1)

    F = array([integrate(lambda x: myfun(x)*x**i, 0,1)[0] for i in range(n)])
```

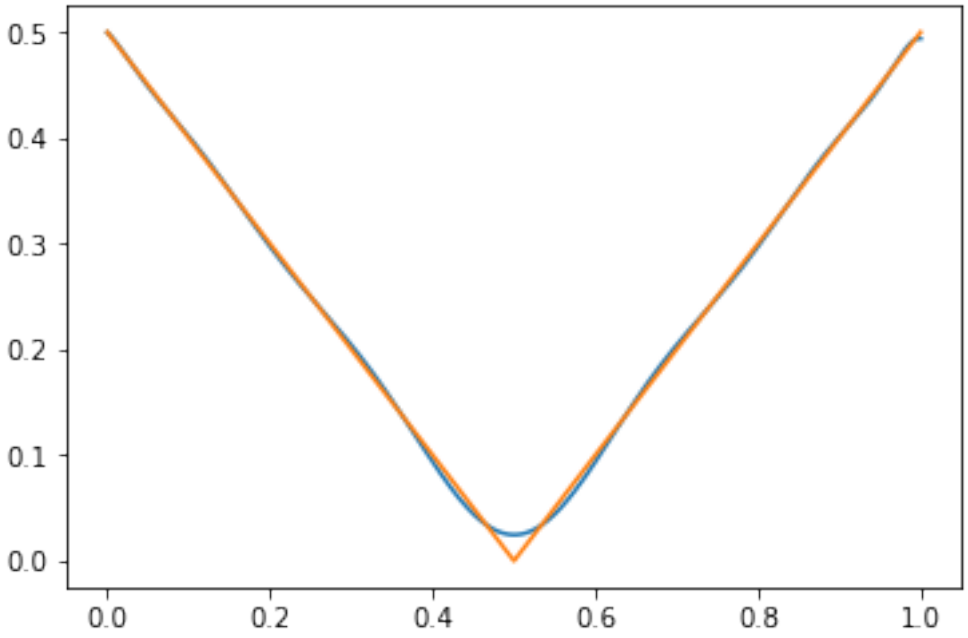
```
    pi = linalg.solve(M, F)
    p = sum([x**i*pi[i] for i in range(n)], axis=0)
    plot(x,p)
plot(x,myfun(x))
```

[14]: [<matplotlib.lines.Line2D at 0x120c069e8>]



[15]:
```
plot(x,p)
plot(x,myfun(x))
```

[15]: [<matplotlib.lines.Line2D at 0x120a1ca90>]



[16]:
```
max(abs(p-myfun(x)))
```

[16]:
```
0.02462443416821003
```

Why do we get these errors in the L2 projection? The matrix M is not well conditioned...

[17]:
```
linalg.cond(M)
```

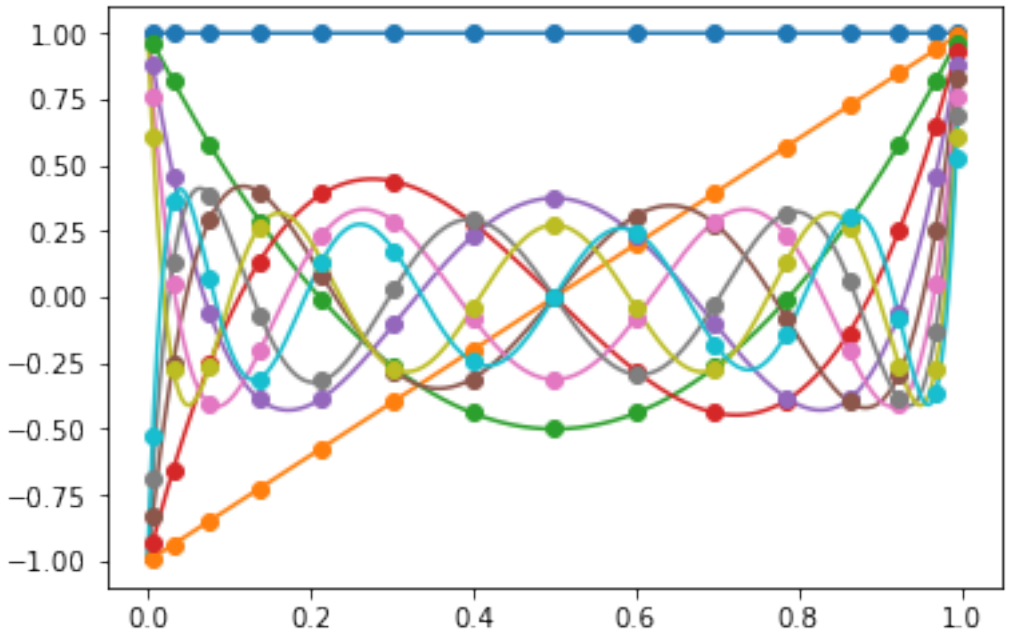
[17]:
```
6.2007862631614438e+17
```

Let's turn to numerical quadrature, and Legendre polynomials (for which M is the identity by construction...)

[18]:
```
from numpy.polynomial.legendre import leggauss
from numpy.polynomial.legendre import legval
from numpy.polynomial.legendre import Legendre
```

[19]:
```
n = 10
N = n+5


q,w = leggauss(N)
w *= .5
q +=1
q /=2
```

[20]:
```
v = array([Legendre.basis(i, domain=[0,1])(x) for i in range(n)])
vq = array([Legendre.basis(i, domain=[0,1])(q) for i in range(n)])
_ = plot(x,v.T)
_ = plot(q, vq.T, 'o')
```



Check that we get a diagonal matrix as M:

[21]:
```
vq.shape
```

[21]:
```
(10, 15)
```

[22]:
```
M = einsum('iq, jq, q', vq, vq, w)
```

[23]:
```
diag = array([M[i,i] for i in range(n)])
```

[24]:
```
diag
```

[24]:
```
array([ 1.        ,  0.33333333,  0.2       ,  0.14285714,  0.11111111,
        0.09090909,  0.07692308,  0.06666667,  0.05882353,  0.05263158])
```

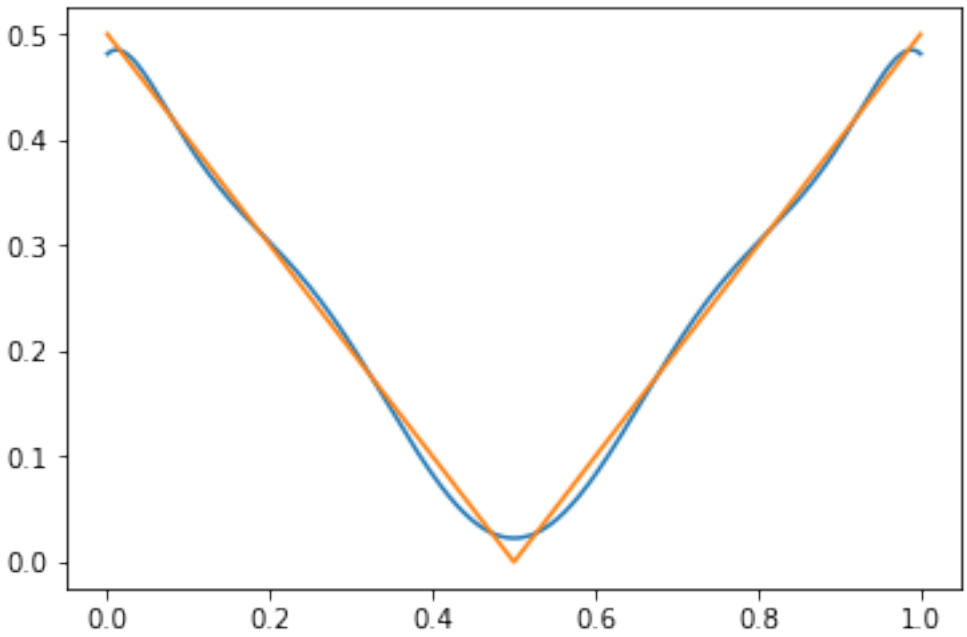
Now perform the integral

[25]:
```
pi = sum(vq*myfun(q)*w, axis=1)
```

And plot the function, and its interpolation

[26]:
```
p = (pi/diag).dot(v)
```

[27]:
```
plot(x, p)
_ = plot(x, myfun(x))
```



[ ]:
```

```

01_interpolation

October 16, 2019

0.1 Lagrange interpolation

Given $(n+1)$ distinct points $\{q_i\}_{i=0}^n$ in the interval $[0,1]$, we define the *Lagrange interpolation* operator $\mathcal{L}^n$ the operator

$$\mathcal{L}^n : C^0([0,1]) \mapsto \mathcal{P}^n$$

which satisfies

$$(\mathcal{L}^n f)(q_i) = f(q_i), \qquad i = 0, \dots, n.$$

This operator is used to approximate the infinitely dimensional space $C^0([0,1])$ with a finite dimensional one, $\mathcal{P}^n$, which is the space of polynomials of order n.

Such a space has dimension $n+1$, and can be constructed using linear combinations of monomials of order $\leq n$:

$$\mathcal{P}^n = \text{span}\{p_i := x^i\}_{i=0}^n$$

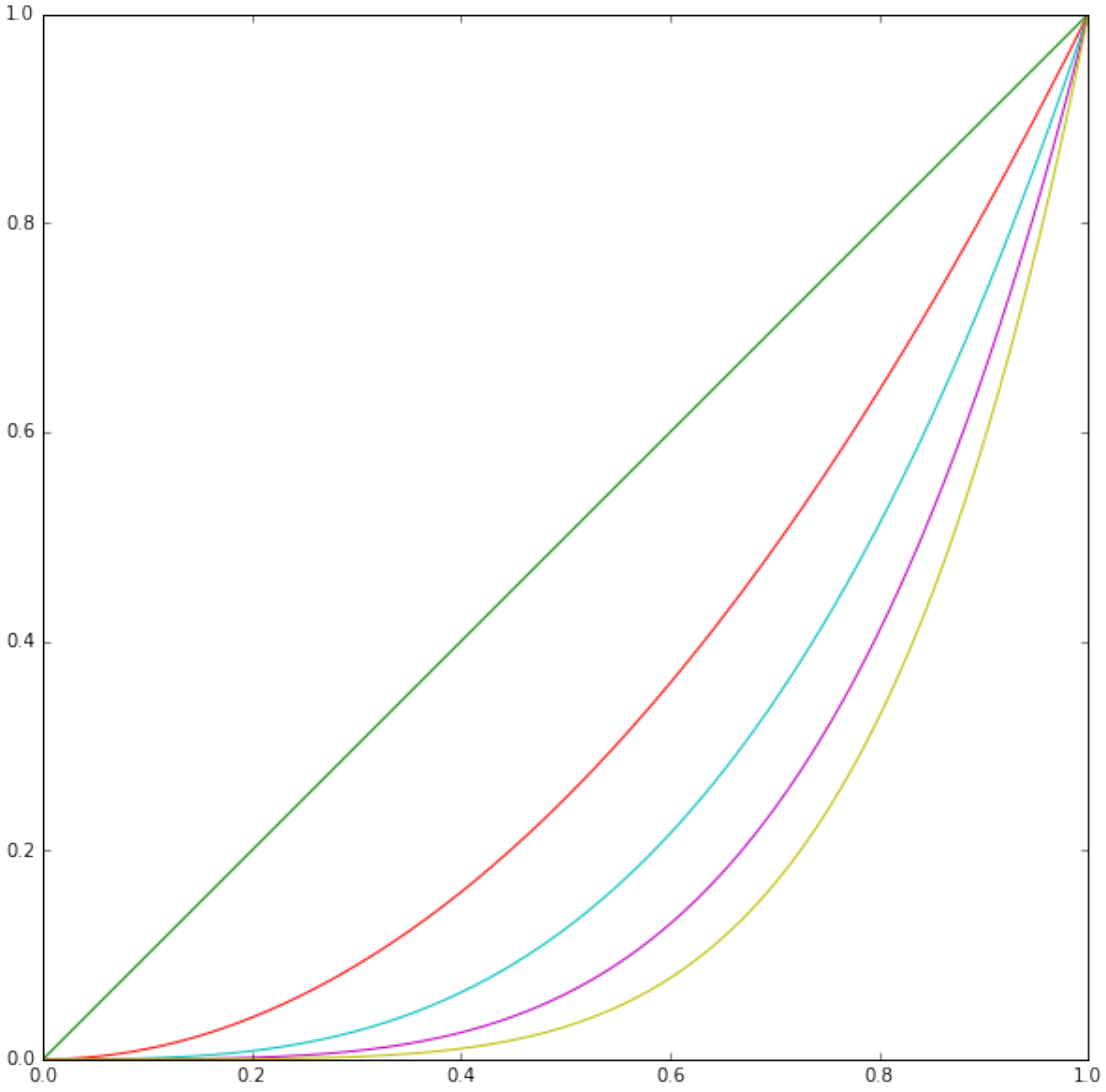
Let's start by importing the usual suspects:

[1]:
```
%matplotlib inline
from numpy import *
from pylab import *
```

In what follows, we will plot several functions in the interval $[0,1]$, so we start by defining a linear space used for plotting. As a good habit, we choose a number of points which would generate intervals that are exactly representable in terms of a binary base.

[2]:
```
ref = 1025 # So that x_i+1 - x_i is exactly representable in base 2
x = linspace(0,1,ref)

n = 5 # Polynomials of order 5, with dimension 6

# We store the basis of Pn row-wise. This is memory efficient if we want to␣
 ↪access
# all the values of the basis at once
Pn = zeros((n+1, len(x)))

for i in xrange(n+1):
    Pn[i] = x**i
```

```
# The _ = in front of the plot command is only there to ignore the output of␣
 ↪the plot
# command
figure(figsize=[10,10])
_ = plot(x, Pn.T)
```



If we want to construct the Lagrange interpolation of a given function on $n+1$ equispaced points in $[0,1]$, then we are actively looking for an element of $\mathcal{P}^n$ that coincides with the function at these given points.

Given a basis $\{p_i\}_{i=0}^n$, any element of $\mathcal{P}^n$ can be written as a linear combination of the basis, i.e.,

$$\forall u \in \mathcal{P}^n, \quad \exists! \{u^i\}_{i=0}^n \quad | \quad u(x) = \sum_{i=0}^{n} u^i p_i(x)$$

in what follows, we'll use Einstein summation convention, and call u both the function of $\mathcal{P}^n$, or the R^{n+1} vector representing its coefficients.

Remark on the notation

We use upper indices to indicate both "contravariant" coefficients and the *canonical basis of the dual space*, i.e., the linear functionals in $(\mathcal{P}^n)^*$ such that

$$(\mathcal{P}^n)^* := \text{span}\{p^i\}_{i=0}^n \qquad | \qquad p^i(p_j) = \delta^i_j \qquad i, j = 0, \dots, n$$

With this notation, we have that the coefficients of a polynomial are uniquely determined by

$$u^i = p^i(u)$$

where the u on the right hand side is an element of $\mathcal{P}^n$ (not its coefficients).

If we want to solve the interpolation problem above, then we need to find the coefficients u^i of the polynomial u that interpolates f at the points q_i:

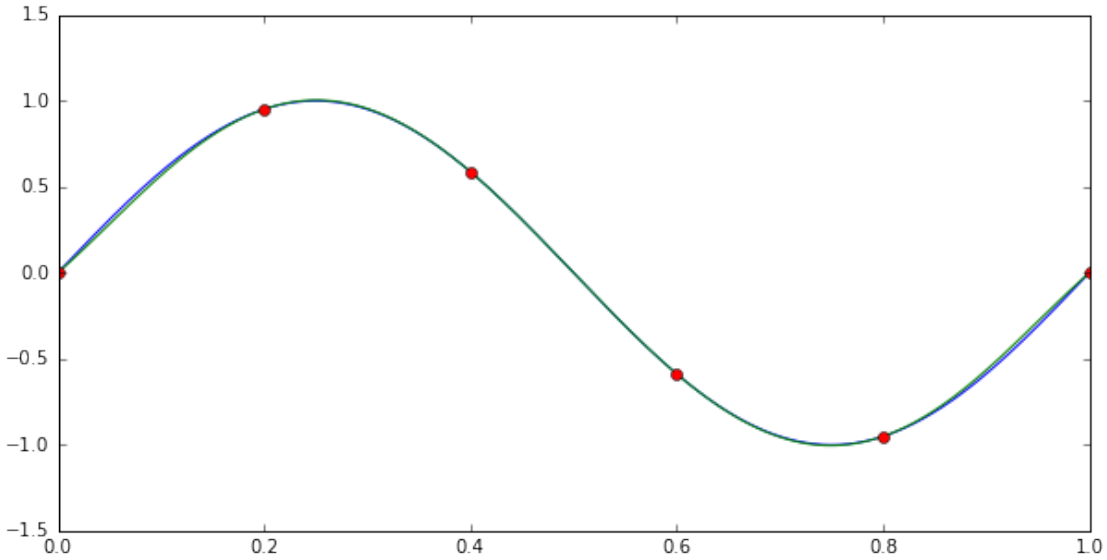
$$p_j(q_i)u^j = f(q_i)$$

(Remember Einstein summation convention)

This can be written as a linear problem $Au = F$, with system matrix $A_{ij} := p_j(q_i)$ and right hand side $F_i = f(q_i)$.

[3]:
```
# The interpolation points
q = linspace(0,1,n+1)

A = zeros((n+1, n+1))
for j in xrange(n+1):
    A[:,j] = q**j

# The interpolation function
f = lambda x: sin(2*pi*x)

# The right hand side
F = f(q)

# The coefficients
u = linalg.solve(A, F)

# Make a nice looking plot
figure(figsize=[10,5])
_ = plot(x, f(x))
```

```
_ = plot(x, Pn.T.dot(u))
_ = plot(q, f(q), 'ro')
```



Is this a good way to proceed with the interpolation? How about the condition number of A? Is it good?

Let's try with an increasing number of points (and degrees):

[4]:
```
for i in xrange(3,15):
    qtmp = linspace(0,1,i)
    Atmp = zeros((i,i))
    for j in xrange(i):
        Atmp[:,j] = qtmp**j

    print("Condition number: (n=", i, ":", linalg.cond(Atmp))
```

```
('Condition number: (n=', 3, ':', 15.099657722502098)
('Condition number: (n=', 4, ':', 98.867738507227671)
('Condition number: (n=', 5, ':', 686.43494181859796)
('Condition number: (n=', 6, ':', 4924.3710566110803)
('Condition number: (n=', 7, ':', 36061.160880212541)
('Condition number: (n=', 8, ':', 267816.70090785547)
('Condition number: (n=', 9, ':', 2009396.3800224287)
('Condition number: (n=', 10, ':', 15193229.677173976)
('Condition number: (n=', 11, ':', 115575244.51733549)
('Condition number: (n=', 12, ':', 883478685.78224337)
('Condition number: (n=', 13, ':', 6780588379.9816332)
('Condition number: (n=', 14, ':', 52214927160.937332)
```

As we see, the condition number of this matrix explodes as n increases. Since the interpolation

problem reduces to solving the matrix constructed as $A_{ij} := p_j(x_i)$, one way to ensure a good condition number is to choose the basis such that A is the identity matrix, i.e., to choose the basis such that $p_j(x_i) = \delta_{ij}$. Such a basis is called the **Lagrange basis**, and it is constructed explicitly as:

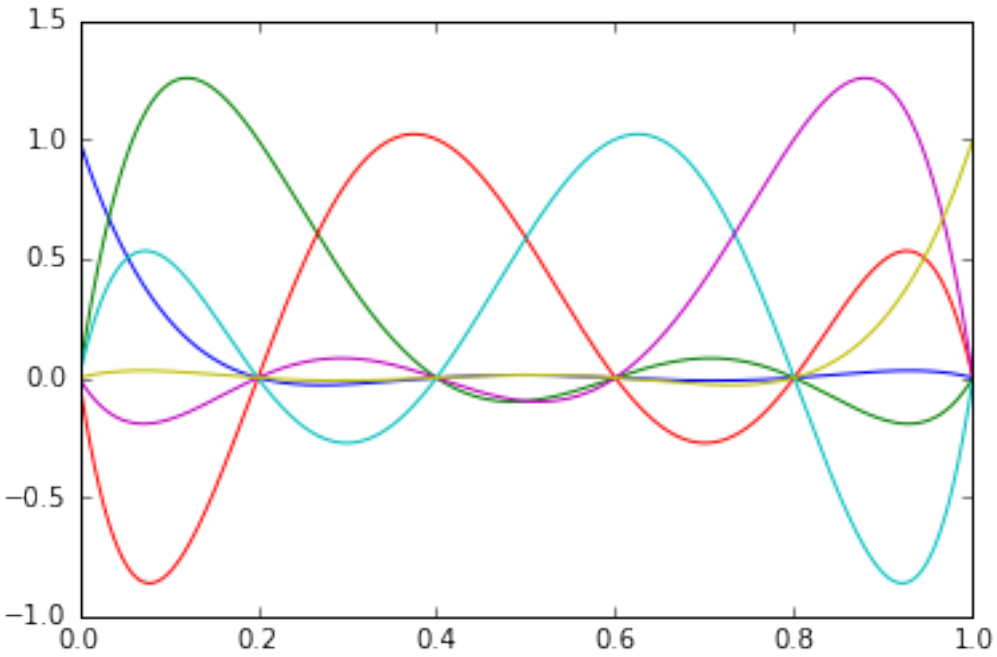
$$l_i^n(x) := \prod_{j=0, j\neq i}^{n} \frac{(x - x_j)}{(x_i - x_j)} \qquad i = 0, \ldots, n$$

With this basis, no matrix inversion is required, and we can simply write the Lagrange interpolation as

$$\mathcal{L}^n f := \sum_{i=0}^{n} f(x_i) l_i^n(x),$$

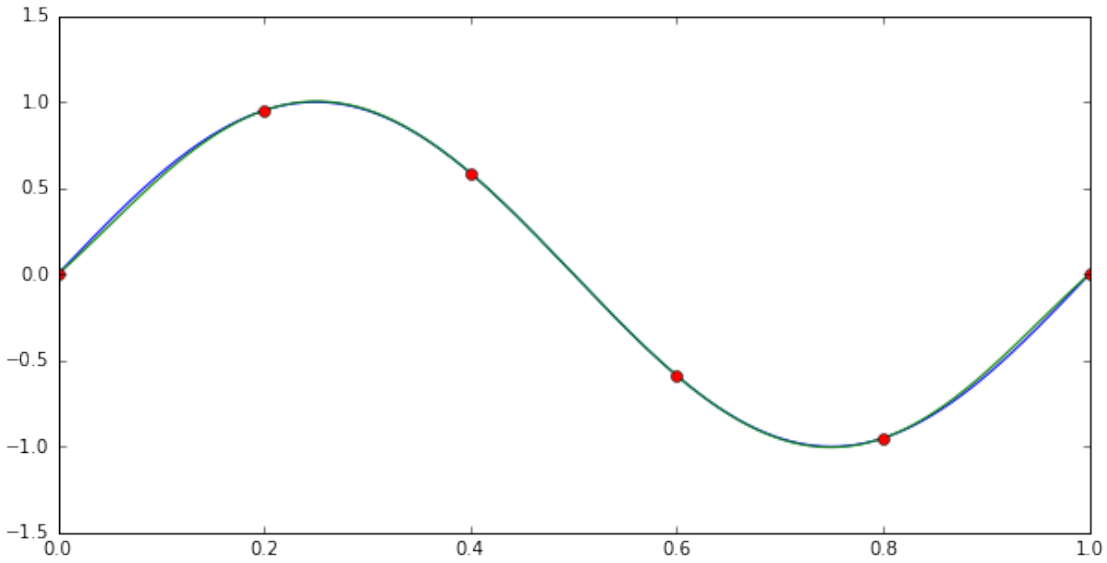
Given a set of $(n+1)$ distinct points $\{x_i\}_{i=0}^n$, there exist a unique Lagrange interpolation of order n.

[5]:
```
Ln = zeros((n+1, len(x)))

for i in xrange(n+1):
    Ln[i] = product([ (x-q[j])/(q[i]-q[j]) for j in xrange(n+1) if j != i],␣
 ↪axis=0)

_ = plot(x,Ln.T)
```



Now the interpolation in the sampling points is simply:

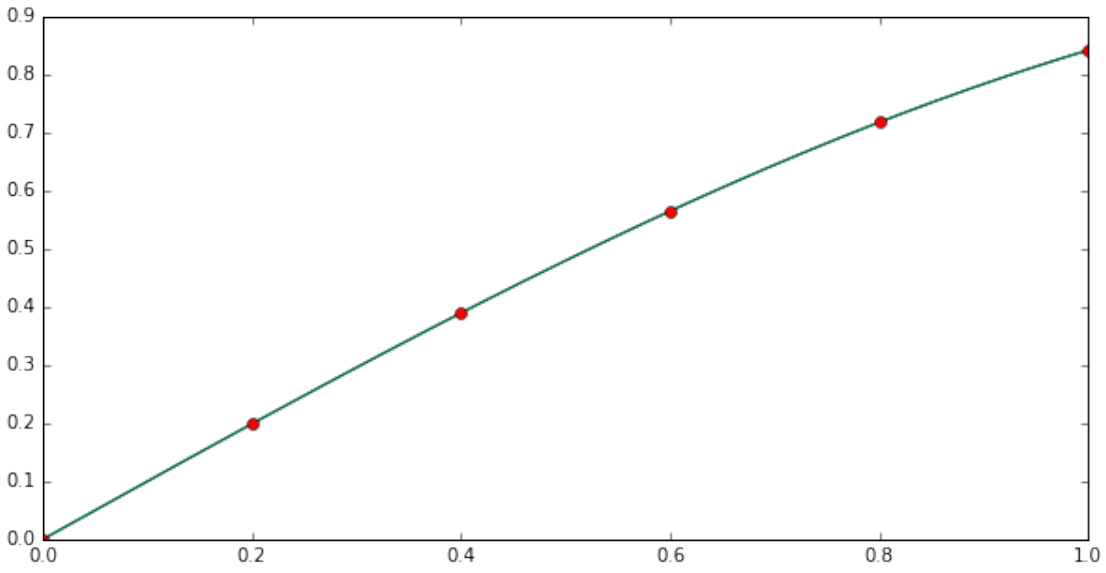
[6]:
```
y = Ln.T.dot(f(q))

figure(figsize=[10,5])
_ = plot(x, f(x))
_ = plot(x, y)
_ = plot(q, f(q), 'ro')
```

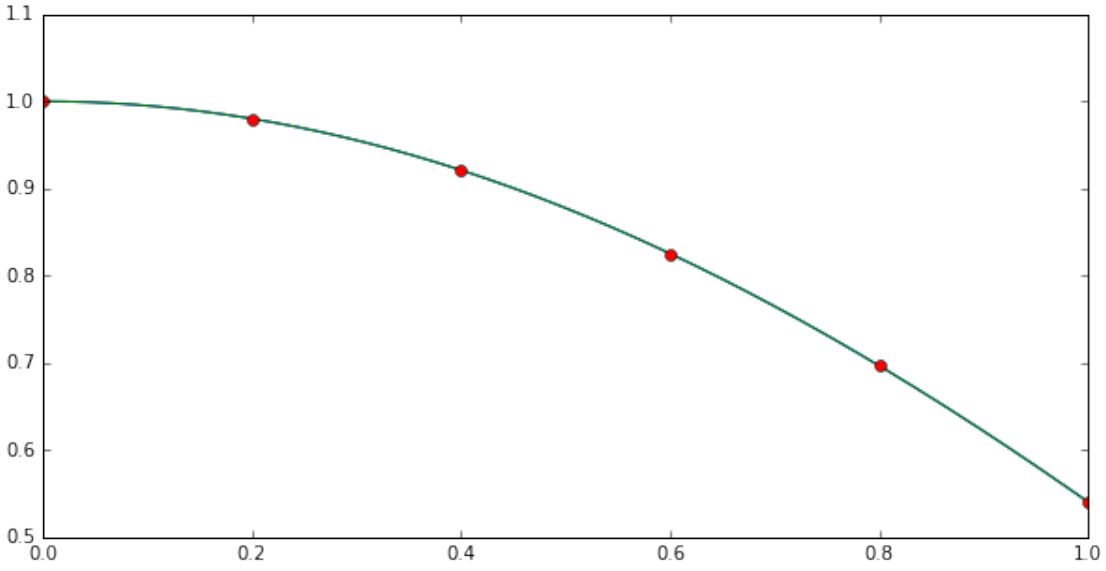


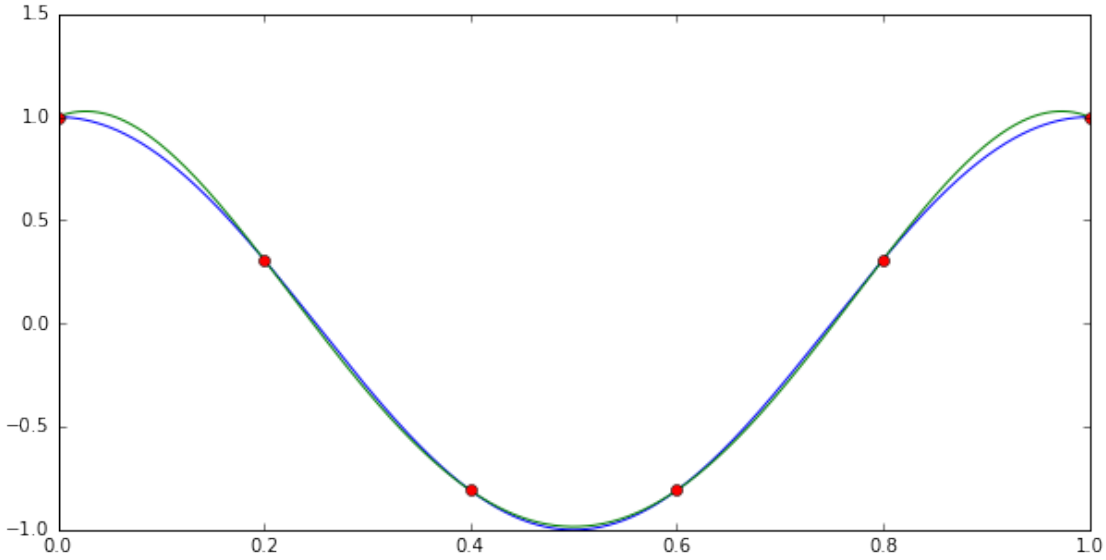
Let's try different functions:

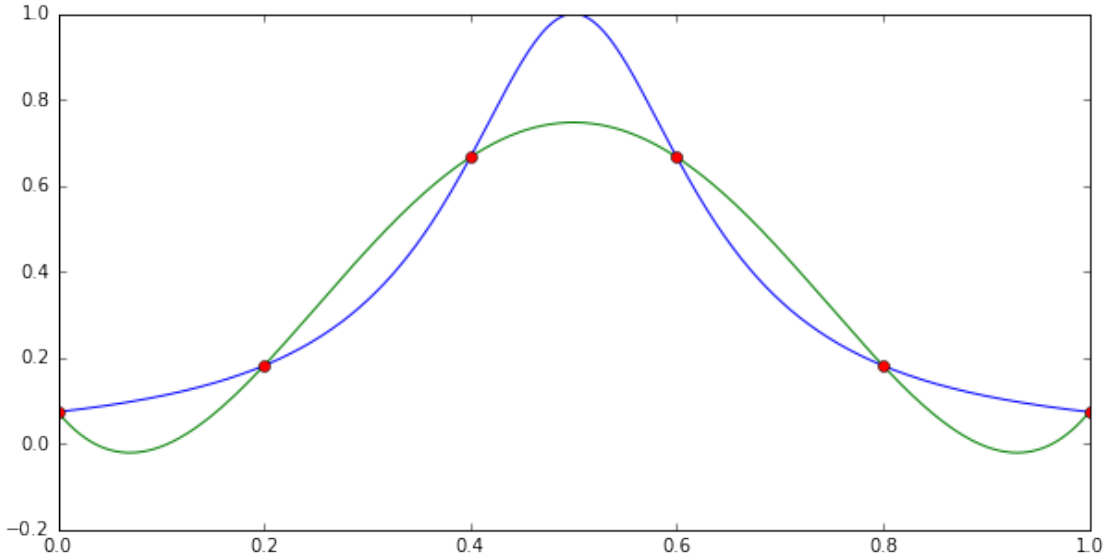
[7]:
```
# A little "macro". This assumes Ln, q, and x are all defined
# Notice: technically this is a python function. However, it
# expects symbols and variables to be defined in the global scope
# and this is **not** good programming style. It may be very
# useful and fast at times, but try not to overdo it.
#
# I'd call it a function if internally it did not use any globally
# defined variable.

def my_plot(f):
    figure(figsize=[10,5])
    y = Ln.T.dot(f(q))
    _ = plot(x, f(x))
    _ = plot(x, y)
    _ = plot(q, f(q), 'ro')
    show()

my_plot(sin)
my_plot(cos)
```

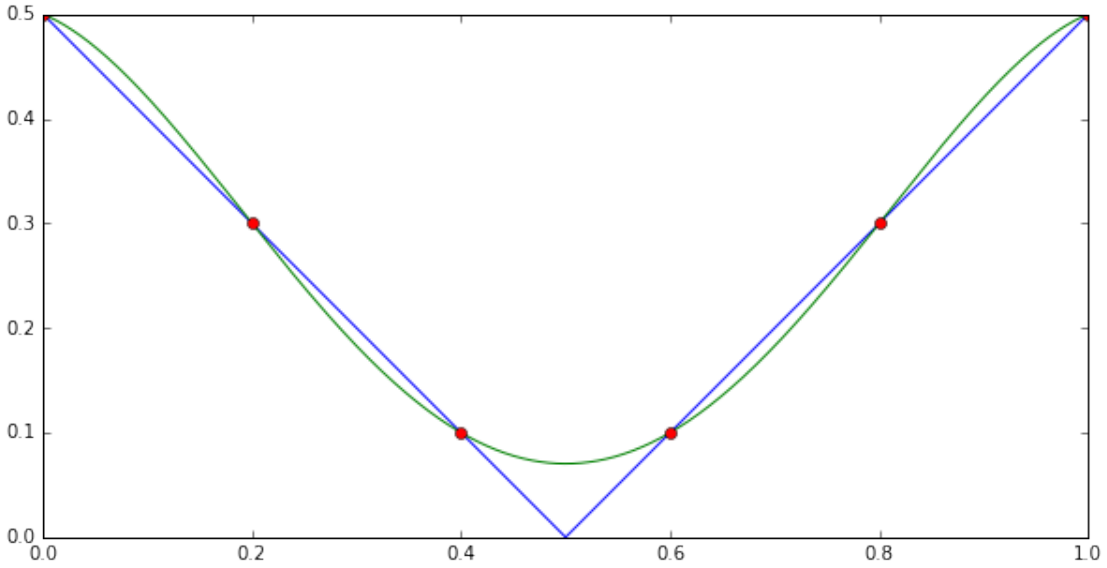
```
# When we need something more complicated than simply cos, or sin,
# we can use "on the fly" function definitions, or lambda functions:
my_plot(lambda x: cos(2*pi*x))

# Lambda functions can be assigned too, for convenience and later
# reuse...
runge = lambda x: 1.0/(1+50*(.5-x)**2)
my_plot(runge)

# Alternatively, you can define the function in the classical pythonic
# way:
def shifted_abs(x):
    return abs(x-.5)

my_plot(shifted_abs)
```




1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.0
0.2
0.4
0.6
0.8
1.0


1.5
1.0
0.5
0.0
−0.5
−1.0
0.0
0.2
0.4
0.6
0.8
1.0


1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0.0
0.2
0.4
0.6
0.8
1.0


0.5
0.4
0.3
0.2
0.1
0.0
0.0
0.2
0.4
0.6
0.8
1.0

[ ]:

03_error_estimation

October 16, 2019

[1]:
```
%matplotlib inline
import numpy as np
import pylab as pl
import sympy as sym
from sympy.functions import Abs
#from sympy import Abs, Symbol, S
```

0.1 Goals of today:

- Check how good or bad are the estimates given in the theoretical lecture
- Compute errors, plot error tables
- Compare Equispaced with Chebyshev

0.2 Lagrangian part

The estimate we want to check:

$$||f - p|| \leq ||f^{n+1}||_\infty \frac{||w(x)||_\infty}{(n+1)!}$$

in order to do so we need to define, simbolic and numerical functions. `Sympy` is a very useful package to handle symbolic expressions and to export them to numerical functions. A the beginnnig of this notebook it is imported with the command: `import sympy as sym`.

Let's start by defining a way to compute the $\|\cdot\|_\infty$ norm, in an approximate way, using numpy.

We use an approximate way which is based on the computation of the l^∞ norm on large n-dimensional vectors, that we use to evaluate and plot our functions.

Begin by defining a linear space, used to evaluate our functions.

[2]:
```
# Using directly numpy broadcasting and max function
l_infty = lambda y: abs(y).max()

# Using full python lists and ranges
def l_infty_1(y):
    m = -1.0
    for i in range(len(y)):
```

```
        m = max(m, abs(y[i]))
    return m

# Iterating over numpy array entries
def l_infty_2(y):
    m = -1.0
    for i in y:
        m = max(m, abs(i))
    return m

# Using numpy norm function
l_infty_3 = lambda y: np.linalg.norm(y, ord=np.inf)

# Test it on a random vector of a million elements
yy = np.random.rand(int(1e6))

%timeit l_infty(yy)
%timeit l_infty_1(yy)
%timeit l_infty_2(yy)
%timeit l_infty_3(yy)

# The timings show that the manual numpy solution is the most efficient.
# The first version and the last do the exact same thing,
# with the difference that the last has some overheads due to
# parsing of the optional parameters, which is not there in the first version
```

```
The slowest run took 4.53 times longer than the fastest. This could mean that an
intermediate result is being cached.
1000 loops, best of 3: 1.01 ms per loop
1 loop, best of 3: 315 ms per loop
1 loop, best of 3: 250 ms per loop
1000 loops, best of 3: 1.02 ms per loop
```

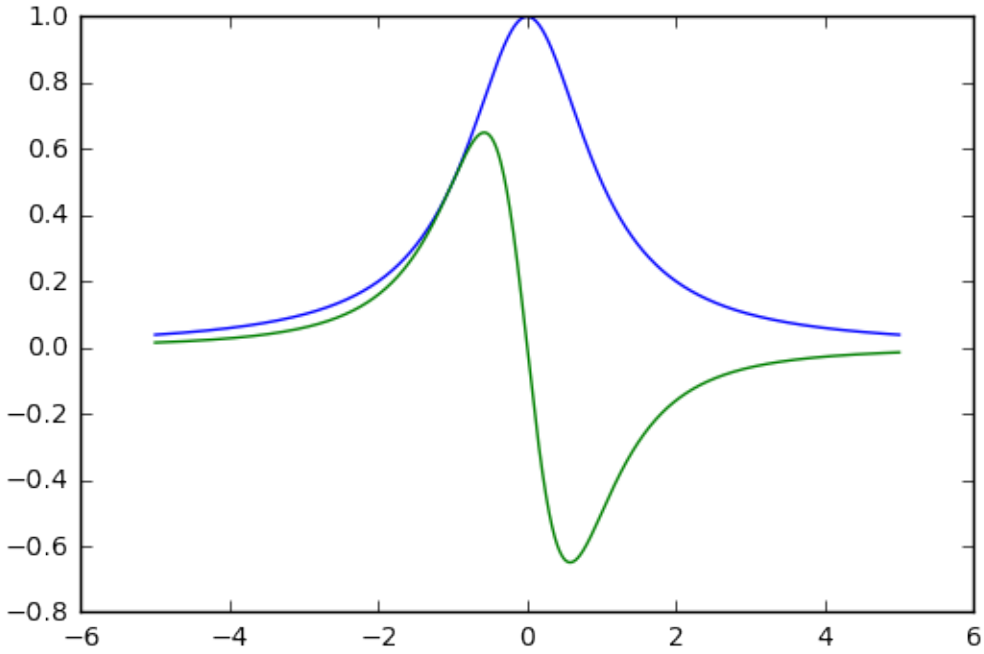
In order to compute derivatives and evaluate symbolic functions, we use sympy. Let's construct a symbolic variable, and define functions in terms of it:

[3]:
```
# Now construct a symbolic function...
t = sym.var('t')
fs = 1.0/(1.0+t**2) # Runge function

fs.diff(t, 1)
```

[3]:
```
-2.0*t/(t**2 + 1.0)**2
```

To make this function *digestible* by numpy we use the simple command `nf = sym.lambdify(t,f, 'numpy')`. This allows the function nf to be called with numpy arguments.

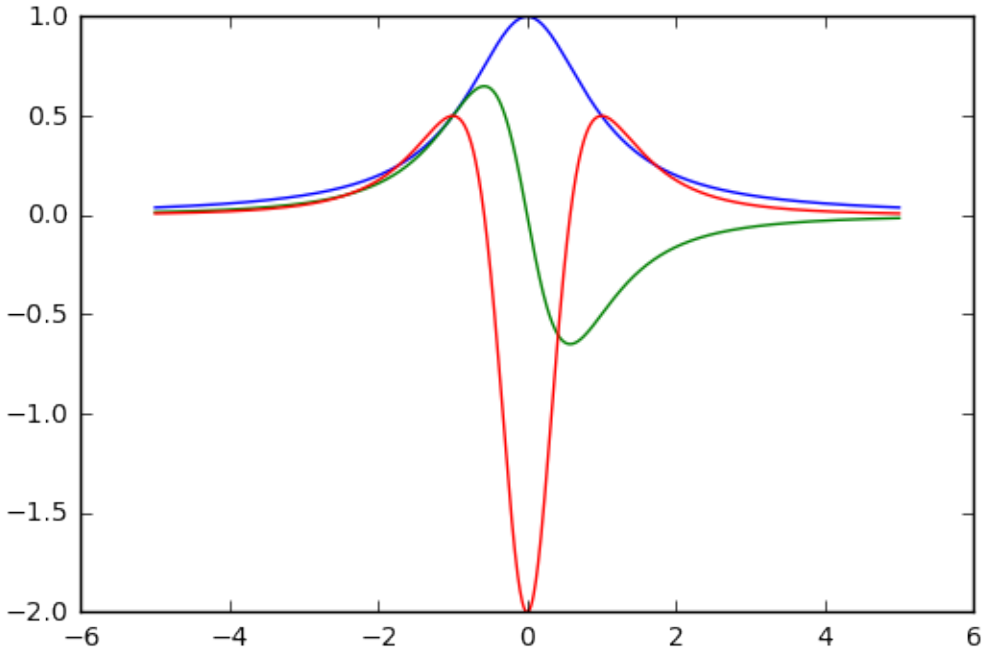
[4]:
```
# Create a vector for the evaluation of functions, norms, etc.
x = np.linspace(-5,5, 2**10+1)

# This won't work
# ns(x)

nf = sym.lambdify(t, fs, 'numpy')
nfprime = sym.lambdify(t, fs.diff(t,1), 'numpy')

# Now we can plot and evaluate the function on numpy arrays
_ = pl.plot(x,nf(x))
_ = pl.plot(x,nfprime(x))
```



Now we construct a small helper function that given a symbolic expression with one single free symbol, it returns a numpy aware lambda function of its n-th derivative, that can be used to evaluate numpy expressions.

[5]:
```
def der(f,n):
    assert len(f.free_symbols) == 1, "I can only do this for single variable␣
 ↪functions..."
    t = f.free_symbols.pop()
    return sym.lambdify(t, f.diff(t, n), 'numpy')

f = der(fs, 0)
fp = der(fs, 1)
```

```
fpp = der(fs, 2)

# Stack columns and plot them all together
_ = pl.plot(x, np.c_[f(x), fp(x), fpp(x)])
```



[6]:
```
# Check derivatives for two functions...
function_set = [fs, sym.sin(2*sym.pi*t)]

for my_f in function_set:
    print("****************")
    print(my_f)
    for i in range(5):
        print(l_infty(der(my_f,i)(x)))
```

```
****************
1.0/(t**2 + 1.0)
1.0
0.649517020733
2.0
4.66826090562
24.0
****************
sin(2*pi*t)
1.0
6.28318530718
```

```
39.4784176044
248.050213442
1558.54545654
```

We aim at controlling all of the pieces of the inequality above, plot how terms behave with the degree, and see what happens :)

Good thing is to start from the beginning and control the term $||f - p||_\infty$. We recall that:

$$p = \mathcal{L}^n f := \sum_{i=0}^{n} f(x_i) l_i^n(x),$$

with

$$l_i^n(x) := \prod_{j=0, j\neq i}^{n} \frac{(x - x_j)}{(x_i - x_j)} \qquad i = 0, \dots, n.$$

Let's implment this guy. We want to fill the matrix `Ln` with $n+1$ rows and as many colums as the number of points where we evaluate the funtion.

$$\mathrm{Ln}_{ij} := l_i(x_j)$$

so that

$$\mathrm{Ln}_{ij} f(q_i) = \sum_{i=0}^{n} l_i(x_j) f(q_i) = (\mathcal{L}^n f)(x_j)$$

A good idea would be to collect the few operations in a function, like this one:

```
def lagrangian_interpolation_matrix(x,q):
    ...
    return Ln
```

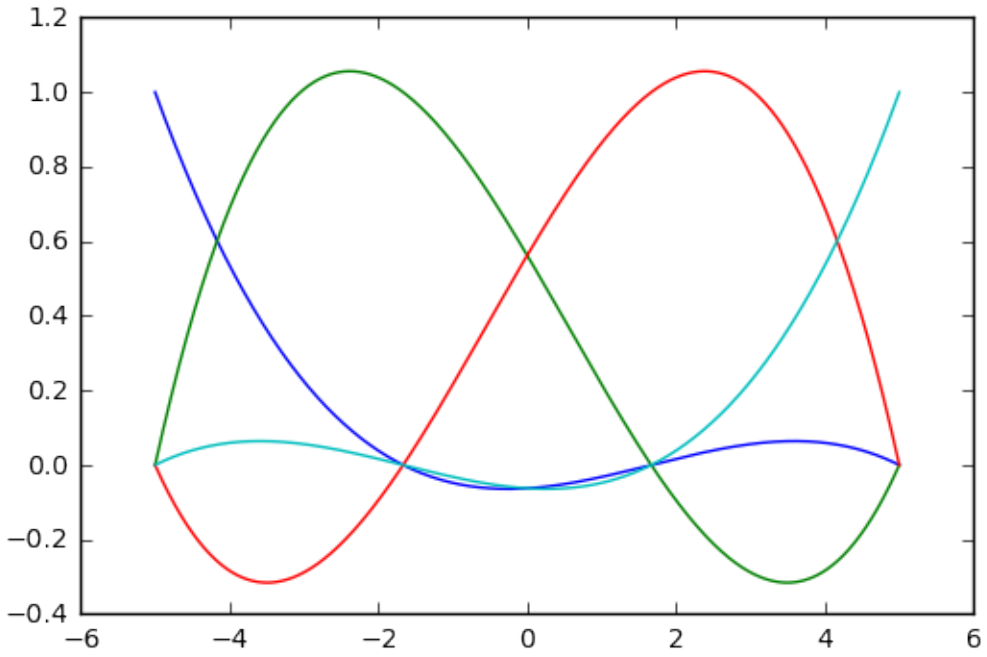
so that we can recall it whenever we need it.

Hint: I wouldn't call myself a good programmer, but I do my best to be like that. First construct the code in the main section of your program, run it, check that it works, then collect the precious commmands you wrote in an function.

0.2.1 Step 0

[7]:
```
n = 3
q = np.linspace(-5,5,n+1)

Ln = np.zeros((n+1, len(x)))

for i in range(n+1):
    Ln[i] = np.ones_like(x)
```

```
    for j in range(n+1):
        if j != i:
            Ln[i] *= (x-q[j])/(q[i]-q[j])

    # Alternative one-liner...
    # Ln[i] = np.product([(x-q[j])/(q[i]-q[j]) for j in range(n+1) if j != i],␣
↪axis=0)

_ = pl.plot(x, Ln.T)
```



0.3 Step 1

Now we transform this into two function that takes the points where we want to compute the matrix, and the interpolation points we use to define the basis.

[8]:
```
def lagrangian_interpolation_matrix(x,q):
    Ln = np.zeros((len(q), len(x)))

    for i in range(len(q)):
        Ln[i] = np.ones_like(x)
        for j in range(len(q)):
            if j != i:
                Ln[i] *= (x-q[j])/(q[i]-q[j])
    return Ln
```

```
def lagrangian_interpolation_matrix_one_liner(x,q):
    Ln = np.zeros((len(q), len(x)))
    for i in range(len(q)):
        Ln[i] = np.product([(x-q[j])/(q[i]-q[j]) for j in range(len(q)) if j !=␣
 ↪i], axis=0)
    return Ln
```

[9]:
```
Error = lagrangian_interpolation_matrix(x,q) -␣
 ↪lagrangian_interpolation_matrix_one_liner(x,q)

print("Error:", np.linalg.norm(Error))
```

```
Error: 0.0
```

From the previous lecture we know that the mathemathical expression:

$$(\mathcal{L}^n f)(x_i) := \sum_{j=0}^{n} f(q_j) l_j^n(x_i) = (\mathrm{Ln}^T f)_i$$

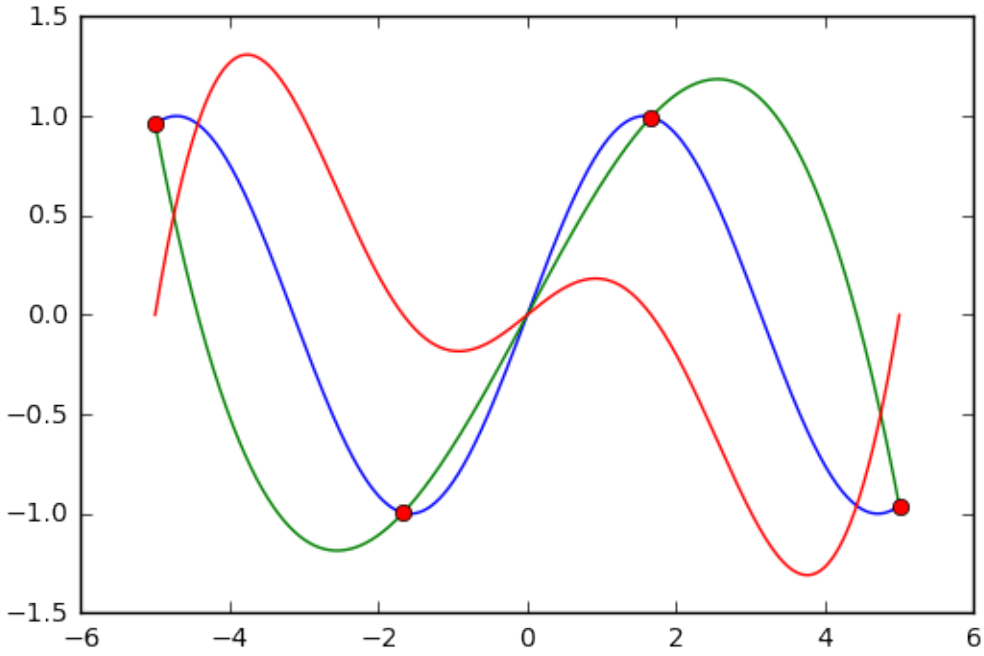
Can be easyly translated into the `numpy` line:

`Ln.T.dot(f(x))`

Let's give it a try:

[10]:
```
fs = sym.sin(t)

f = der(fs,0)

n = 3
q = np.linspace(-5,5,n+1)

Ln = lagrangian_interpolation_matrix(x,q)

_ = pl.plot(x, f(x))
_ = pl.plot(x, Ln.T.dot(f(q)))
_ = pl.plot(q, f(q), 'ro')

e = f(x) - Ln.T.dot(f(q))
_ = pl.plot(x, e)

Error = l_infty(e)

print("Error:", Error)
```

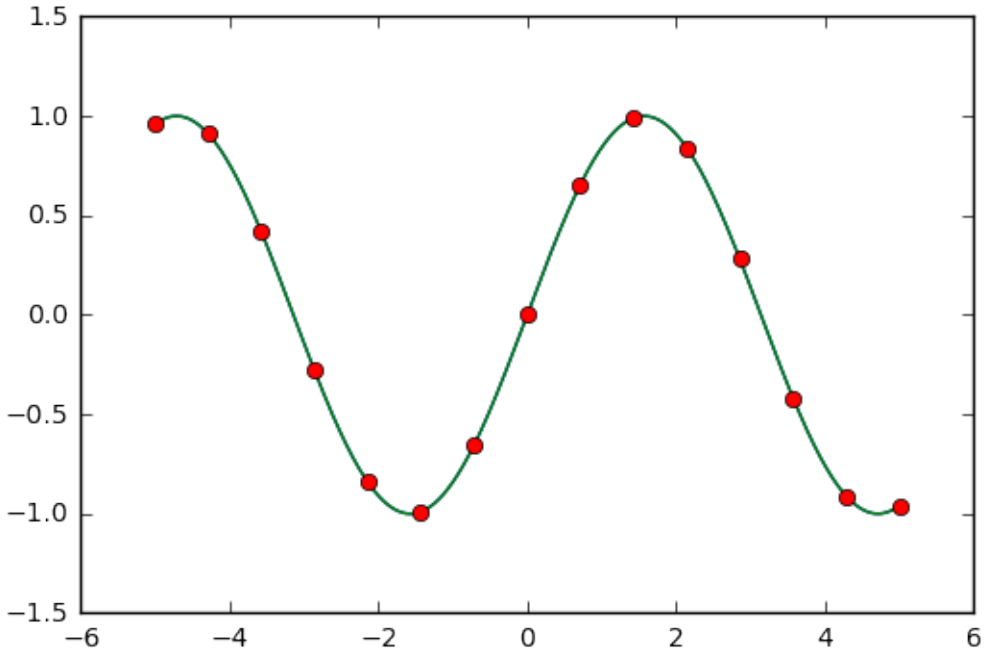
```
Error: 1.30879781308
```



Let's increase the number of points...

[11]:
```
q = np.linspace(-5,5,15)
Ln = lagrangian_interpolation_matrix(x,q)

_ = pl.plot(x, f(x))
_ = pl.plot(x, Ln.T.dot(f(q)))
_ = pl.plot(q, f(q), 'ro')


e = f(x) - Ln.T.dot(f(q))

Error = l_infty(e)

print("Error:", Error)
```

```
Error: 3.16643288822e-05
```



Now compute the following

$$||f - p||_\infty =$$

`error`

$$||f^{n+1}||_\infty =$$

`nth_der`

$$w(x) = \prod_{i=0}^{n}(x - q_i), \quad ||w(x)||_\infty =$$

`w`

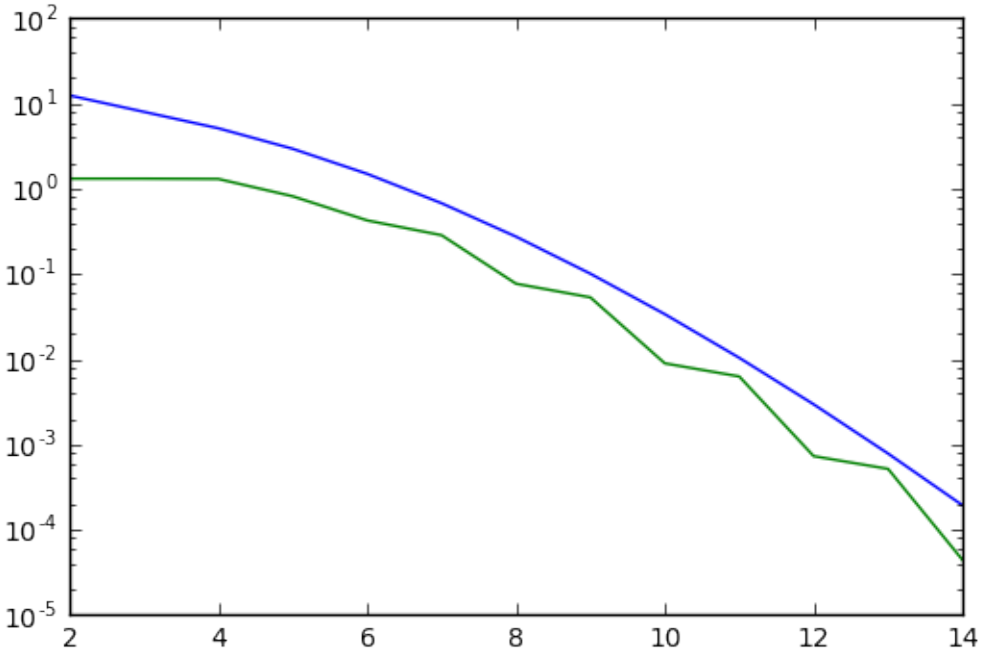
[12]:
```
# define w
w = lambda x,q: np.product([x-q[i] for i in range(len(q))], axis=0)

q = np.linspace(-5,5,5)

Ln = lagrangian_interpolation_matrix(x,q)
error = l_infty(f(x) - Ln.T.dot(f(q)))

fs = sym.sin(2*sym.pi*t)
fp = der(fs, len(q))
```

```
nth_der = l_infty(fp(x))
w_infty = l_infty(w(x,q))

UpperEstimate = nth_der*w_infty/np.math.factorial(len(q))
print(UpperEstimate)
```

```
28939.8354451
```

[13]:
```
fs = sym.sin(t)

points = range(2,15)
UpperEstimate = []
ActualError = []

for n in points:
    q = np.linspace(-5,5,n)

    Ln = lagrangian_interpolation_matrix(x,q)
    ActualError.append(l_infty(f(x) - Ln.T.dot(f(q))))

    fp = der(fs, len(q))
    nth_der = l_infty(fp(x))

    w_infty = l_infty(w(x,q))

    UpperEstimate.append(nth_der*w_infty/np.math.factorial(len(q)))

print(UpperEstimate)

_ = pl.semilogy(points, UpperEstimate)
_ = pl.semilogy(points, ActualError)
```

```
[12.499986507269831, 8.0187320709228516, 5.1439958400563608, 2.9552669407295622,
1.5022989519097527, 0.679306689276586, 0.27558908243703184, 0.10119474297253792,
0.033904686060053868, 0.010436685726412308, 0.0029701957053380654,
0.00078557830648299338, 0.00019410452628791942]
```

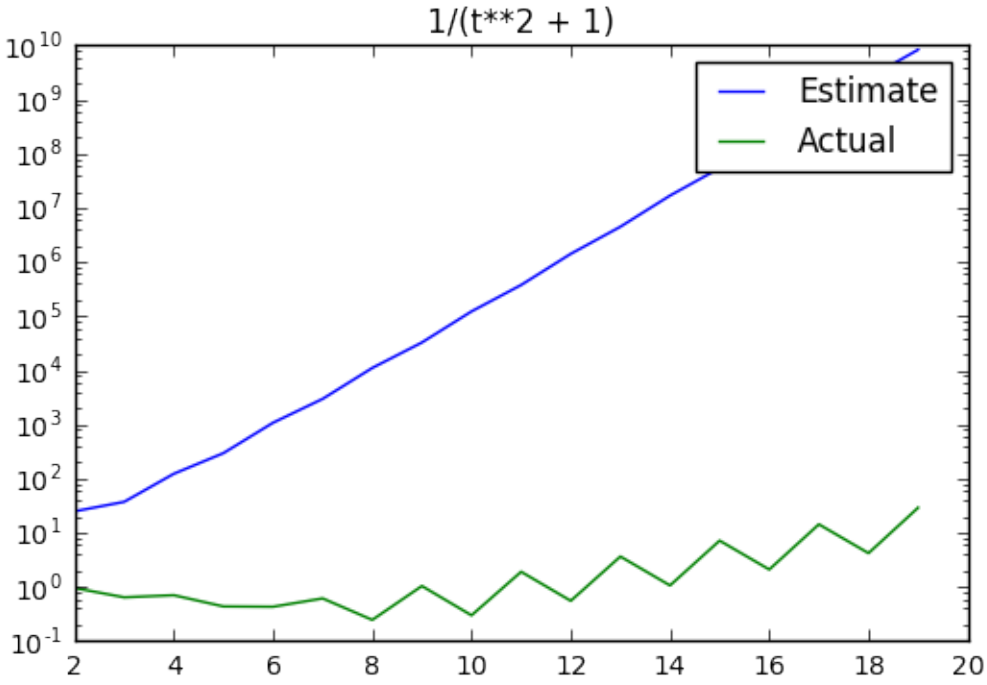


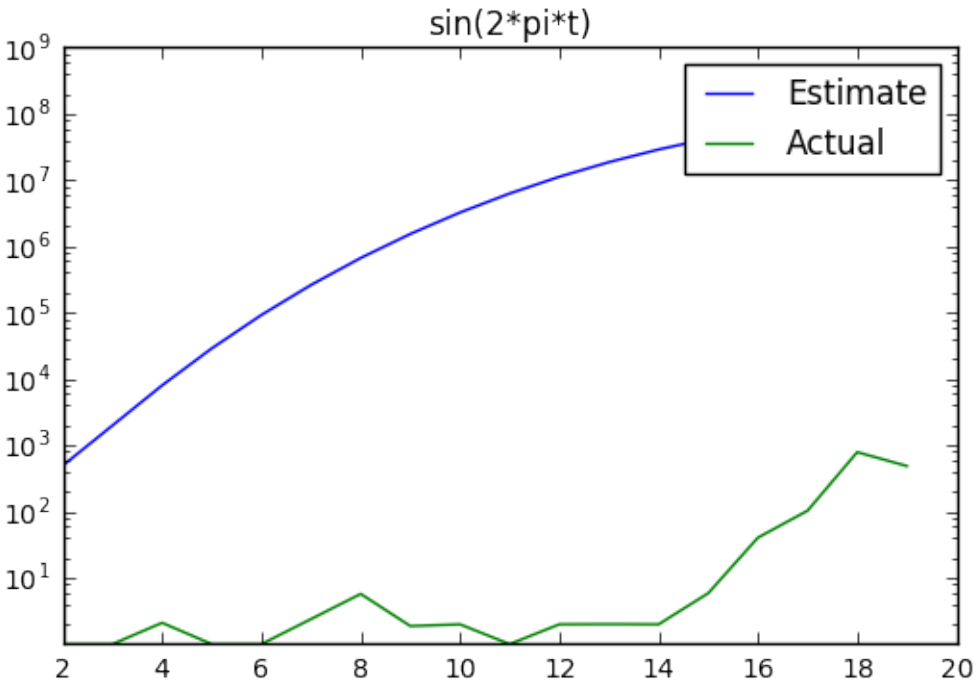
If I didn't mess the code this a good spot to play aroud with the function to be checked. Let's save everything into a single function. Let's also look forward. Instead of using by default equispaced points, lets' ask for a function that can generate the points for us, given x and q...

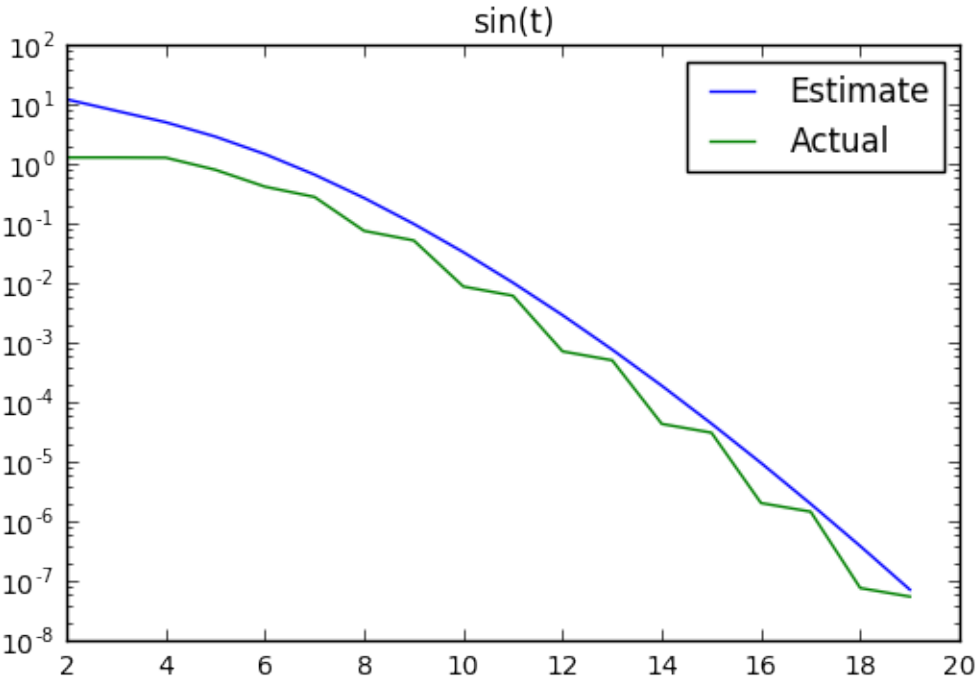
[14]:
```
def check_errors(x, fs, n, generator=lambda x,n: np.linspace(x.min(),x.
↪max(),n)):
    """
    Check the error for the interpolation obtained by calling
    q = generator(x,i)
    for each i in the range(2,n).

    The default generator uses equispaced points.
    """
    points = range(2,n)
    UpperEstimate = []
    ActualError = []

    f = der(fs, 0)

    for n in points:
        q = generator(x,n)

        Ln = lagrangian_interpolation_matrix(x,q)
        ActualError.append(l_infty(f(x) - Ln.T.dot(f(q))))
```

```
        fp = der(fs, len(q))
        nth_der = l_infty(fp(x))

        w_infty = l_infty(w(x,q))

        UpperEstimate.append(nth_der*w_infty/np.math.factorial(len(q)))

    return (points, UpperEstimate, ActualError)
```

[15]:
```
function_set = [1/(1+t**2), sym.sin(2*sym.pi*t), sym.sin(t)]

for fs in function_set:
    p, u, e = check_errors(x, fs, 20)

    _ = pl.semilogy(p, np.c_[u, e])
    pl.title(str(fs))
    pl.legend(['Estimate', 'Actual'])
    pl.show()
```

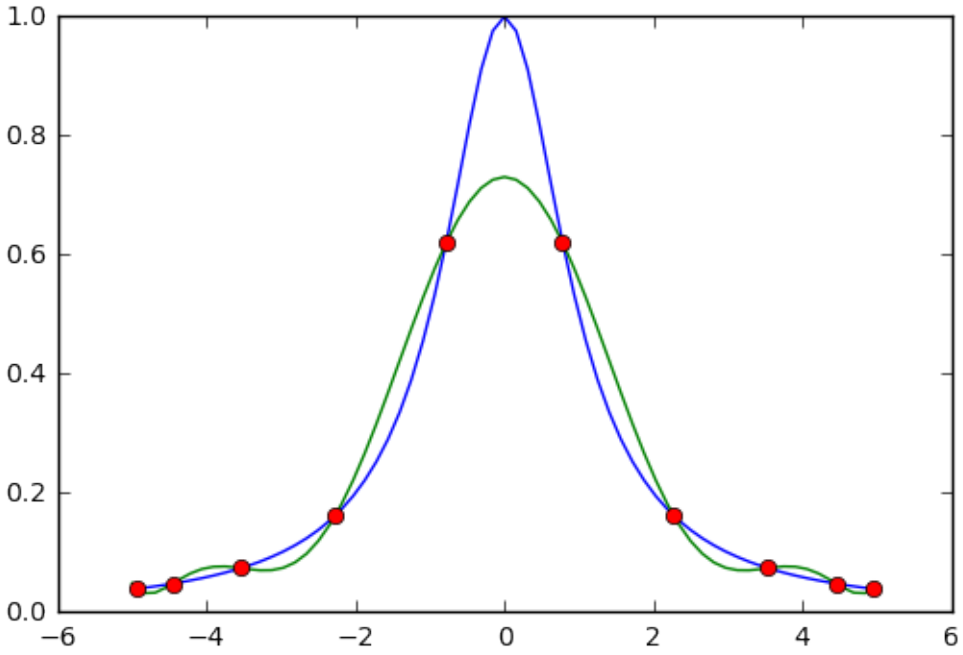






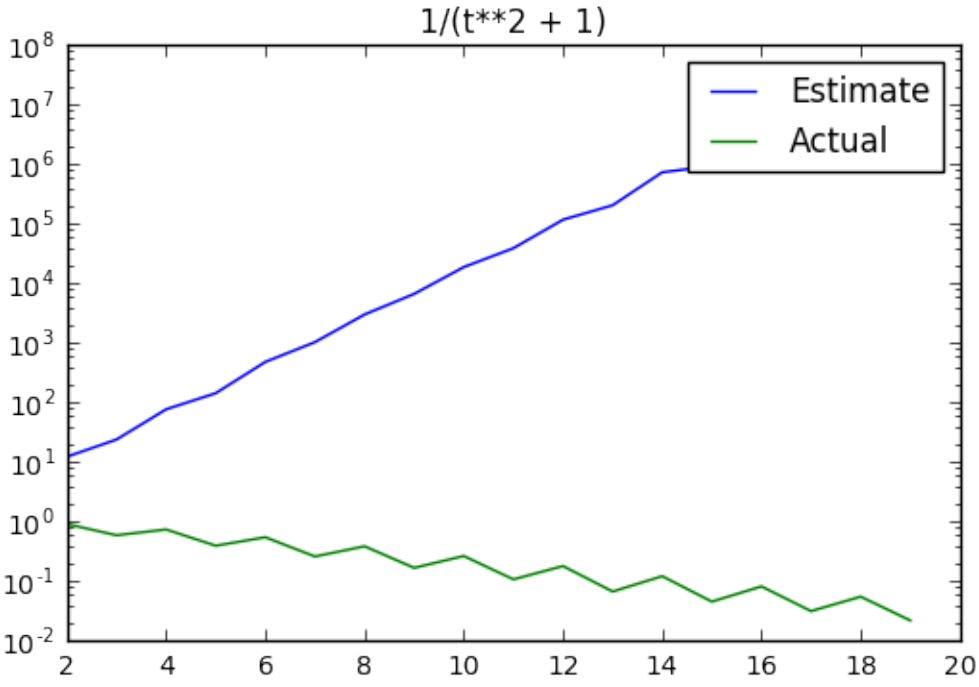
Now let's try to repeat the same thing with **Chebyshev** points...

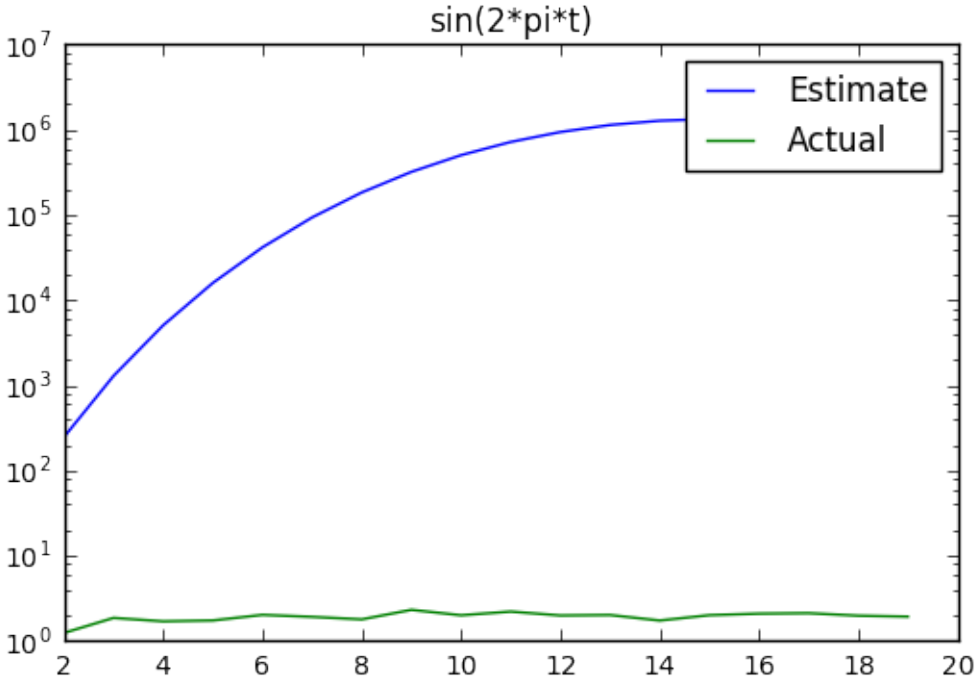
[16]:
```
# We start by trying to interpolate the Runge function:

# Get a smaller sample of points...
x = np.linspace(-5,5,2**6+1)

# This is used to generate chebyshev points between xmin and xmax
generator = lambda x,n: x.min()+(x.max()-x.min())/2*(np.polynomial.chebyshev.
 ↪chebgauss(n)[0]+1)

# 10 interpolation points
q = generator(x,10)

# The the Runge function
f = der(function_set[0], 0)

# The interpolation matrix
Ln = lagrangian_interpolation_matrix(x,q)

_ = pl.plot(x, np.c_[f(x), Ln.T.dot(f(q))])
_ = pl.plot(q, f(q), 'or')
```

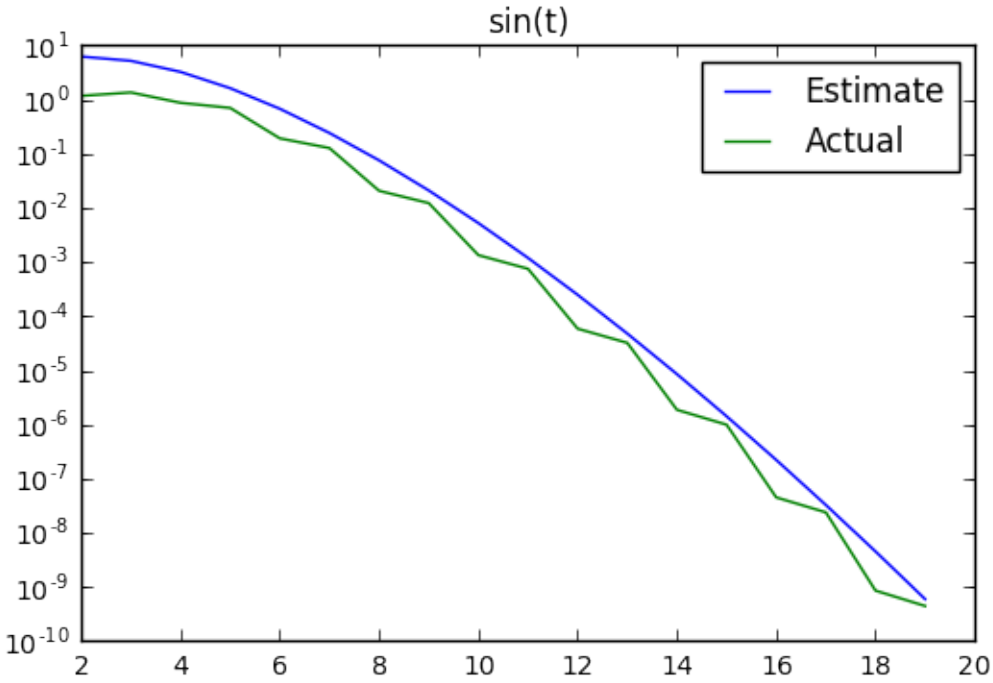


[17]:
```
for fs in function_set:
    p, u, e = check_errors(x, fs, 20, generator)

    _ = pl.semilogy(p, np.c_[u, e])
```

```
    pl.title(str(fs))
    pl.legend(['Estimate', 'Actual'])
    pl.show()
```






sin(t)
Estimate
Actual
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{-9}
10^{-10}
2
4
6
8
10
12
14
16
18
20

04_best_approximation

October 16, 2019

1 Best Approximation in Hilbert Spaces

[1]:
```
%matplotlib inline
import sympy as sym
import pylab as pl
import numpy as np
import numpy.polynomial.polynomial as n_poly
import numpy.polynomial.legendre as leg
```

1.1 Mindflow

We want the best approximation (in Hilbert Spaces) of the function f, on the space $V = \text{span}\{v_i\}$. Remember that $p \in V$ is best approximation of f if and only if:

$$(p - f, q) = 0, \quad \forall q \in V.$$

Focus one second on the fact that both p and q belong to V. We know that any q can be expressed as a linear combination of the basis functions v_i:

$$(p - f, v_i) = 0, \quad \forall v_i \in V.$$

Moreover p is uniquely defined by the coefficients p^j such that $p = p^j\, v_j$. Collecting this information together we get:

$$(v_j, v_i)p^j = (f, v_i), \quad \forall v_i \in V.$$

Now that we know our goal (finding these p^j coefficients) we do what the rangers do: we explore!

We understand that we will need to invert the matrix:

$$M_{ij} = (v_j, v_i) = \int v_i \cdot v_j$$

What happens if we choose basis functions such that $(v_j, v_i) = \delta_{ij}$?

How to construct numerical techniques to evaluate integrals in an efficient way?

Evaluate the L^2 projection.

1.2 Orthogonal Polynomials

Grham Schmidt

$$p_0(x) = 1, \qquad p_k(x) = x^k - \sum_{j=0}^{k-1} \frac{(x^k, p_j(x))}{(p_j(x), p_j(x))}$$

or, alternatively

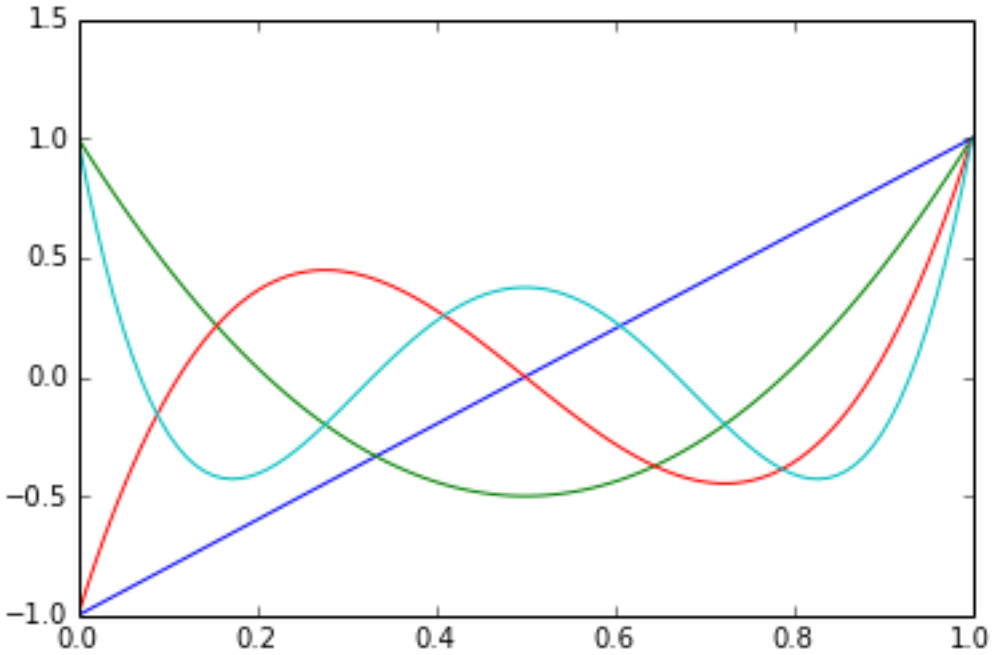
$$p_0(x) = 1, \qquad p_k(x) = x\, p_{k-1}(x) - \sum_{j=0}^{k-1} \frac{(x p_{k-1}(x), p_j(x))}{(p_j(x), p_j(x))}$$

[2]:
```
def scalar_prod(p0,p1,a=0,b=1):
    assert len(p0.free_symbols) <= 1, "I can only do this for single variable␣
 ↪functions..."
    t = p0.free_symbols.pop() if len(p0.free_symbols) == 1 else sym.symbols('t')
    return sym.integrate(p0*p1,(t,a,b))
```

[4]:
```
t = sym.symbols('t')
k = 3

Pk = [1+0*t] # Force it to be a sympy expression

for k in range(1,5):
    s = 0
    for j in range(0,k):
        s+= scalar_prod(t**k,Pk[j])/scalar_prod(Pk[j],Pk[j])*Pk[j]
    pk = t**k-s
    # pk = pk/sym.sqrt(scalar_prod(pk,pk))
    pk = pk/pk.subs(t,1.)
    Pk.append(pk)


M = []
for i in range(len(Pk)):
    row = []
    for j in range(len(Pk)):
        row.append(scalar_prod(Pk[i],Pk[j]))
    M.append(row)

M = sym.Matrix(M)

print(M)
```

```
x = np.linspace(0,1,100)

for p in Pk:
    if p != 1 :
        fs = sym.lambdify(t, p, 'numpy')
        #print x.shape
        #print fs(x)
        _ = pl.plot(x,fs(x))
```

```
Matrix([[1, 0, 0, 1.77635683940025e-15, -5.32907051820075e-15], [0,
0.333333333333333, -8.88178419700125e-16, 0, -7.10542735760100e-15], [0,
-8.88178419700125e-16, 0.200000000000003, 2.84217094304040e-14,
-5.68434188608080e-14], [1.77635683940025e-15, 0, 2.84217094304040e-14,
0.142857142857110, -3.26849658449646e-13], [-5.32907051820075e-15,
-7.10542735760100e-15, -5.68434188608080e-14, -3.26849658449646e-13,
0.111111111112791]])
```



1.3 Theorem

Le q be nonzero polynomial of degree $n+1$ and $\omega(x)$ a positive weight function, s. t.:

$$\int_a^b x^k q(x)\,\omega(x) = 0, \quad k = 0, \ldots, n$$

If x_i are zeros of $q(x)$, then:

$$\int_a^b f(x)\,\omega(x) \approx \sum_{i=0}^{n} w_i\, f(x_i)$$

with:

$$w_i = \int_a^b l_i(x)\,\omega(x)$$

is exact for all polynomials of degree at most $2n+1$. Here $l_i(x)$ are the usual Lagrange interpolation polynomials.

Proof: assume $f(x)$ is a polynomial of degree at most $2n+1$ and show:

$$\int_a^b f(x)\,\omega(x) = \sum_{i=0}^{n} w_i\, f(x_i).$$

Usign the polynomial division we have:

$$\underbrace{f(x)}_{2n+1} = \underbrace{q(x)}_{n+1}\,\underbrace{p(x)}_{n} + \underbrace{r(x)}_{n}.$$

By taking x_i as zeros of $q(x)$ we have:

$$f(x_i) = r(x_i)$$

Now:

$$\begin{aligned}\int_a^b f(x)\,\omega(x) &= \int_a^b [q(x)\,p(x) + r(x)]\,\omega(x) \\ &= \underbrace{\int_a^b q(x)\,p(x)\,\omega(x)}_{=0} + \int_a^b r(x)\,\omega(x)\end{aligned}$$

Since $r(x)$ is a polynomial of order n this is exact:

$$\int_a^b f(x)\,\omega(x) = \int_a^b r(x)\,\omega(x) = \sum_{i=0}^{n} w_i\, r(x_i)$$

But since we chose x_i such that $f(x_i) = r(x_i)$, we have:

$$\int_a^b f(x)\,\omega(x) = \int_a^b r(x)\,\omega(x) = \sum_{i=0}^{n} w_i\, f(x_i)$$

This completes the proof.

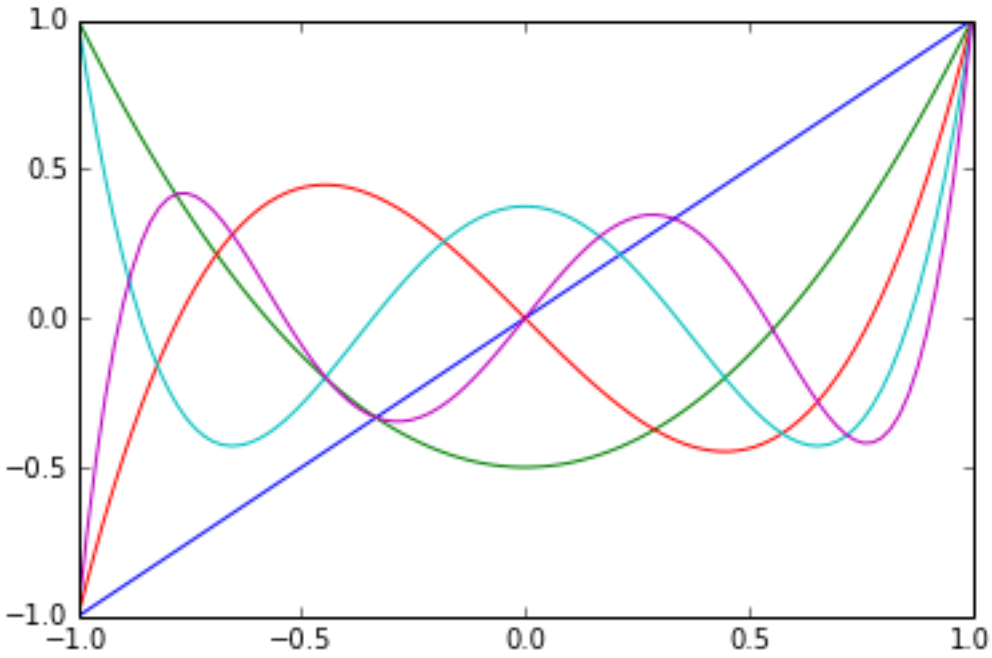
1.4 Legendre Polynomial

Two term recursion, to obtain the same orthogonal polynomials above (defined between [-1,1]), normalized to be one in $x = 1$:

$$(n+1)p^{n+1}(x) = (2n+1)\,x\,p^{n}(x) - n\,p^{n-1}(x)$$

[5]:
```
Pn = [1.,t]

#Pn = [1.,x, ((2*n+1)*x*Pn[n] - n*Pn[n-1])/(n+1.) for n in range(1,2)]

for n in range(1,5):
    pn1 = ((2*n+1)*t*Pn[n] - n*Pn[n-1])/(n+1.)
    Pn.append(sym.simplify(pn1))

print(Pn)

#print(sym.poly(p))
#print(sym.real_roots(sym.poly(p)))
print(sym.integrate(Pn[4]*Pn[3],(t,-1,1)))


x = np.linspace(-1,1,100)
for p in Pn:
    if p != 1. :
        fs = sym.lambdify(t, p, 'numpy')
        #print x.shape
        #print fs(x)
        _ = pl.plot(x,fs(x))
```

```
[1.0, t, 1.5*t**2 - 0.5, t*(2.5*t**2 - 1.5), 4.375*t**4 - 3.75*t**2 + 0.375,
t*(7.875*t**4 - 8.75*t**2 + 1.875)]
0
```



In our proof we selected to evaluate x_i at the zeros of the legendre polynomials, this is why we need to evaluate the zeros of the polynomials.

[6]:
```
print(sym.real_roots(sym.poly(Pn[2])))

#q = [-1.]+sym.real_roots(sym.poly(Pn[2]))+[1.]
q = sym.real_roots(sym.poly(Pn[3]))
print(q)

#for p in Pn:
#    if p != 1. :
#        #print(sym.poly(p))
#        #print(sym.real_roots(sym.poly(p)))
#        print(sym.nroots(sym.poly(p)))
```

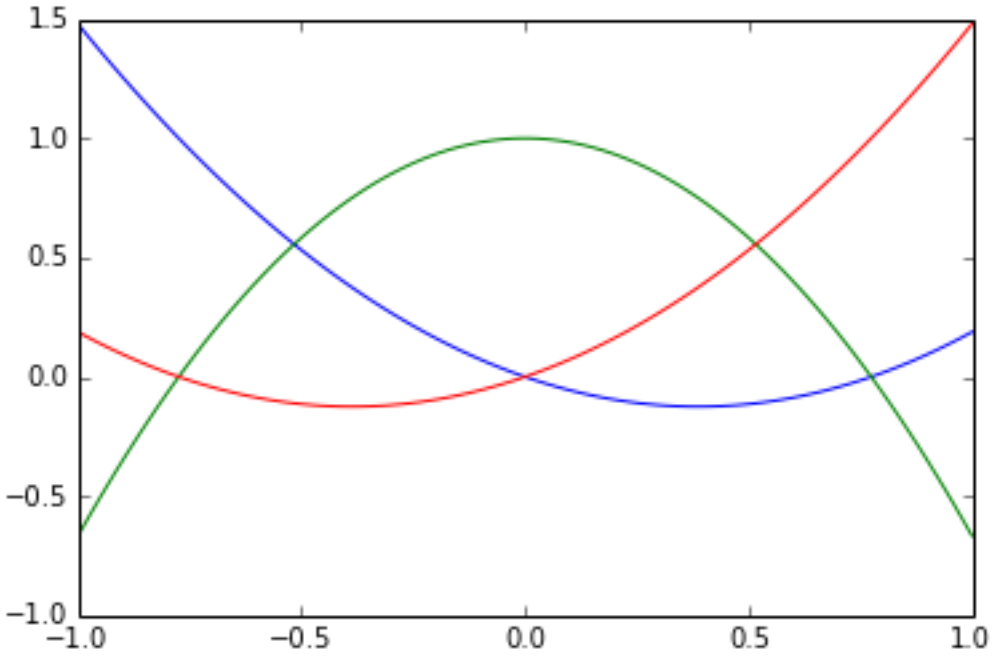
```
[-sqrt(3)/3, sqrt(3)/3]
[-sqrt(15)/5, 0, sqrt(15)/5]
```

$$w_i = \int_{-1}^{1} l_i(x)$$

[7]:
```
Lg = [1. for i in range(len(q))]
print(Lg)

#for i in range(n+1):
```

```
for i in range(len(q)):
    for j in range(len(q)):
        if j != i:
            Lg[i] *= (t-q[j])/(q[i]-q[j])

print(Lg)

x = np.linspace(-1,1,100)

for l in Lg:
    fs = sym.lambdify(t, l, 'numpy')
    _ = pl.plot(x,fs(x))
```

```
[1.0, 1.0, 1.0]
[0.833333333333333*t*(t - sqrt(15)/5), -1.66666666666667*(t - sqrt(15)/5)*(t +
sqrt(15)/5), 0.833333333333333*t*(t + sqrt(15)/5)]
```



[8]:
```
for l in Lg:
    print(sym.integrate(l,(t,-1,1)))
```

```
0.555555555555555
0.888888888888889
0.555555555555555
```

1.4.1 Hint

Proiezione usando polinomi LEGENDRE (f,v_i)

2 Now let's get Numerical

From now on I work on the $[0, 1]$ interval, becouse i like it this way :)

In the previus section we explored what sympbolically was happening, now we implement things on the computer. We saw how important are the legendre plynomials. Here a little documentation on that. I pont it out not because you need to read it all, but because I would like you get some aquitance with this criptic documentation pages doc.

The problem we aim at solving is finding the coefficents p_j such that:

$$(v_j, v_i)p^j = (f, v_i), \quad \forall v_i \in V.$$

Remind in this section the einstein notation holds.

We can expand the compact scalar product notation:

$$p^j \int_0^1 v_i \, v_j = \int_0^1 f \, v_i, \quad \forall v_i \in V.$$

We consider $V = \text{span}\{l_i\}$. Our problem becomes:

$$p^j \int_0^1 l_i \, l_j = \int_0^1 f \, l_i, \quad \text{for } i = 0, \dots, \texttt{deg}$$

Let's focus on mass matrix:

$$\int_0^1 l_i(x) \, l_j(x) = \sum_k l_i(x_k) \, w_k \, l_j(x_k) =$$

$$= \begin{pmatrix} l_0(x_0) & l_0(x_1) & \dots & l_0(x_q) \\ l_1(x_0) & l_1(x_1) & \dots & l_1(x_q) \\ & \dots & \dots & \\ l_n(x_0) & l_n(x_1) & \dots & l_n(x_q) \end{pmatrix} \begin{pmatrix} w_0 & 0 & \dots & 0 \\ 0 & w_1 & \dots & 0 \\ & \dots & \dots & \\ 0 & 0 & \dots & w_q \end{pmatrix} \begin{pmatrix} l_0(x_0) & l_1(x_0) & \dots & l_n(x_0) \\ l_0(x_1) & l_1(x_1) & \dots & l_n(x_1) \\ & \dots & \dots & \\ l_0(x_q) & l_1(x_q) & \dots & l_n(x_q) \end{pmatrix} = B \, W \, B^T$$

A piece of curiosity, how the the two functions to find theros in two different ways

[10]:
```
print sym.nroots(sym.poly(Pn[-1]))
coeffs = np.zeros(6)
coeffs[-1] = 1.
print(leg.legroots(coeffs))
```

```
[-0.906179845938664, -0.538469310105683, 0, 0.538469310105683,
0.906179845938664]
[ -9.06179846e-01  -5.38469310e-01  -5.96500148e-17   5.38469310e-01
   9.06179846e-01]
```

[11]:
```
print gauss_points(3)
print(np.sqrt(3./5.)*.5)+.5
```

```
[ 0.11270167  0.5         0.88729833]
0.887298334621
```

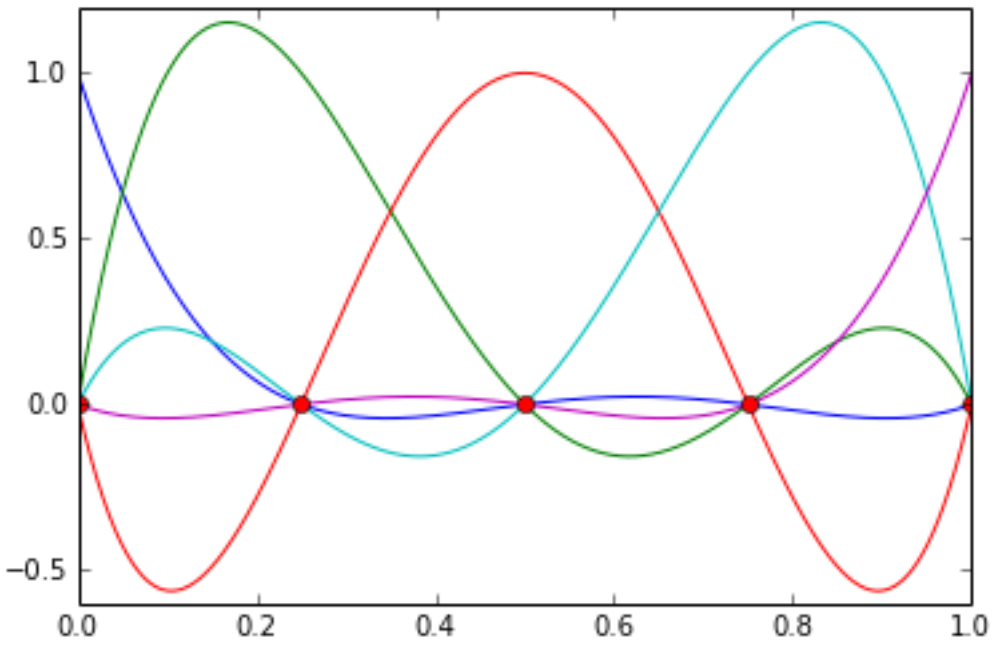
[12]:
```
def define_lagrange_basis_set(q):
    n = q.shape[0]
    L = [n_poly.Polynomial.fromroots([xj for xj in q if xj != q[i]]) for i in␣
 ↪range(n)]
    L = [L[i]/L[i](q[i]) for i in range(n)]
    return L
```

differenza fra le roots ”simboliche” e non

[14]:
```
deg = 4
Nq = deg+1
p,w = leg.leggauss(Nq)
w = .5 * w
p = .5*(p+1)
#print p
#print w
W = np.diag(w)
#print W
```

[15]:
```
int_p = np.linspace(0,1,deg+1)
L = define_lagrange_basis_set(int_p)

print(len(L))

x = np.linspace(0,1,1025)
for f in L:
    _ = pl.plot(x, f(x))
_ = pl.plot(int_p, 0*int_p, 'ro')
```

```
5
```



[16]:
```
B = np.zeros((0,Nq))
for l in L:
    B = np.vstack([B,l(p)])
```

Recall:

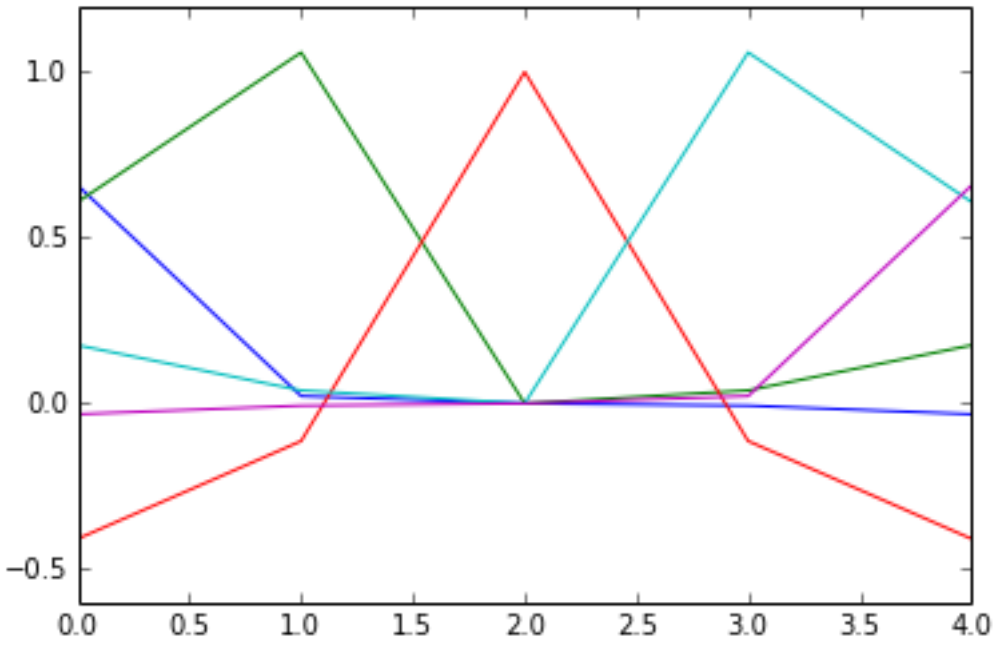
$$B\,W\,B^T p = BWf$$

$$B\,W\,B^T = \begin{pmatrix} l_0(x_0) & l_0(x_1) & \dots & l_0(x_q) \\ l_1(x_0) & l_1(x_1) & \dots & l_1(x_q) \\ & & \ddots & \\ l_n(x_0) & l_n(x_1) & \dots & l_n(x_q) \end{pmatrix} \begin{pmatrix} w_0 & 0 & \dots & 0 \\ 0 & w_1 & \dots & 0 \\ & & \ddots & \\ 0 & 0 & \dots & w_q \end{pmatrix} \begin{pmatrix} l_0(x_0) & l_1(x_0) & \dots & l_n(x_0) \\ l_0(x_1) & l_1(x_1) & \dots & l_n(x_1) \\ & & \ddots & \\ l_0(x_q) & l_1(x_q) & \dots & l_n(x_q) \end{pmatrix}$$

[17]:
```
print(B.shape)
_ = pl.plot(B.T)
M = B.dot(W.dot(B.T))
print np.linalg.matrix_rank(M)
print np.linalg.cond(M)
```

```
(5, 5)
[[  6.57727883e-01   2.20631033e-02  -8.88178420e-16  -6.61878610e-03
   -3.23726701e-02]
 [  6.07692695e-01   1.05879718e+00   1.77635684e-15   3.92223408e-02
    1.75534108e-01]
 [ -4.08582015e-01  -1.13463840e-01   1.00000000e+00  -1.13463840e-01
```

```
   -4.08582015e-01]
 [  1.75534108e-01   3.92223408e-02   4.44089210e-16   1.05879718e+00
    6.07692695e-01]
 [ -3.23726701e-02  -6.61878610e-03  -2.22044605e-16   2.20631033e-02
    6.57727883e-01]]
5
14.1390607069
```



[18]:
```
def step_function():
    def sf(x):
        index = where((x>.3) & (x<.7))
        step = zeros(x.shape)
        step[index] = 1
        return step
    return lambda x : sf(x)
```

$$BWf = \begin{pmatrix} l_0(x_0) & l_0(x_1) & \dots & l_0(x_q) \\ l_1(x_0) & l_1(x_1) & \dots & l_1(x_q) \\ & \dots & \dots & \\ l_n(x_0) & l_n(x_1) & \dots & l_n(x_q) \end{pmatrix} \begin{pmatrix} w_0 & 0 & \dots & 0 \\ 0 & w_1 & \dots & 0 \\ & \dots & \dots & \\ 0 & 0 & \dots & w_q \end{pmatrix} \begin{pmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_q) \end{pmatrix}$$

[19]:
```
g = lambda x: np.sin(2*np.pi*x)
#g = step_function()
p = p.reshape((p.shape[0],1))
```

```
G = g(p)
print G.shape
print B.shape
print W.shape
G = B.dot(W.dot(G))
```

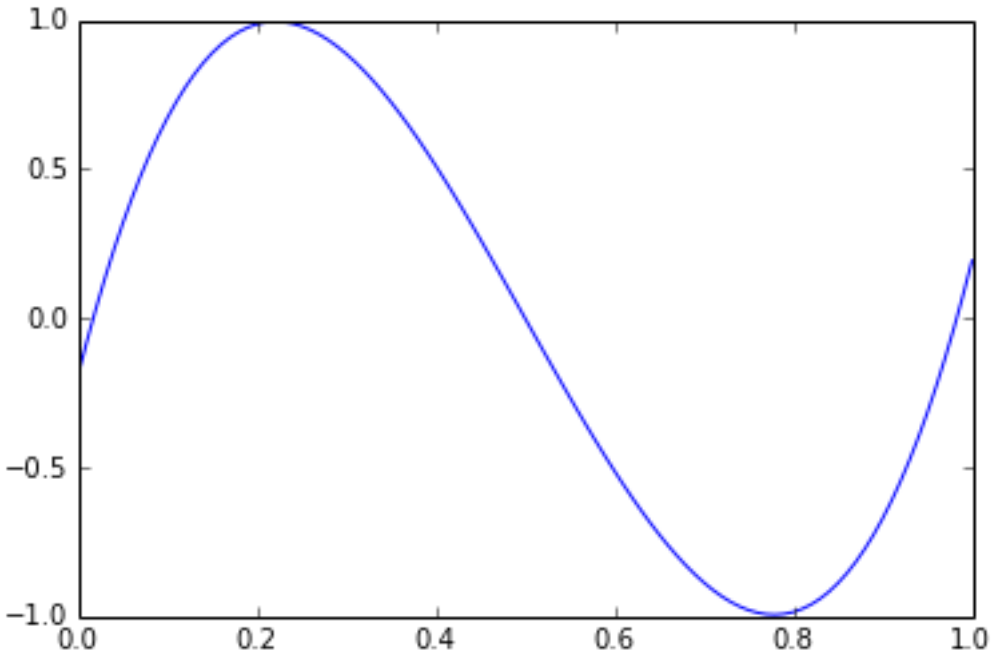
```
(5, 1)
(5, 5)
(5, 5)
```

[20]:
```
u = np.linalg.solve(M, G)
print u
```

```
[[ -1.92161045e-01]
 [  9.79052672e-01]
 [ -1.35712301e-15]
 [ -9.79052672e-01]
 [  1.92161045e-01]]
```

[21]:
```
def get_interpolating_function(LL,ui):
    def func(LL,ui,x):
        acc = 0
        for L,u in zip(LL,ui):
            #print(L,u)
            acc+=u*L(x)
        return acc
    return lambda x : func(LL,ui,x)
```

[22]:
```
I = get_interpolating_function(L,u)
sampling = np.linspace(0,1,101)
_= pl.plot(sampling, I(sampling))
#plot(xp, G,'ro')
```



2.1 Diference in between projection and interpolation runge example

Proiezione usando polinomi LEGENDRE (f,v_i) con quadratura con 18 punti

Interpolazione usando polinomi LAGRANGE (sui punti di quadratura che sono i punti di gauss della funzione sopra)

[ ]:

05a_linear_systems_direct

October 16, 2019

1 Direct methods for solving linear systems

Recall the prototypal PDE problem introduce in the Lecture 08:

$$-u_{xx}(x) = f(x) \quad \text{in } \Omega = (0,1)$$

$$u(x) = 0, \quad \text{on } \partial\Omega = \{0,1\}$$

The physical interpretation of this problem is related to the modelling of an elastic string, which occupies at rest the space $[0,1]$ and is fixed at the two extremes. The unknown $u(x)$ represents the displacement of the string at the point x, and the right-hand side models a prescribed force $f(x)$ on the string.

For the numerical discretization of the problem, we consider a **Finite Difference (FD) Approximation**. Let n be an integer, a consider a uniform subdivision of the interval $(0,1)$ using n equispaced points, denoted by $\{x_i\}_{i=0}^n$. Moreover, let u_i be the FD approximation of $u(x_i)$, and similarly $f_i \approx f(x_i)$.

In order to formulate the discrete problem, we consider a FD approximation of the left-hand side, as follows:

$$-u_{xx}(x_i) \approx \frac{-u_{i-1} + 2u_i - u_{i+1}}{h^2}$$

being $h = \frac{1}{n-1}$ the size of each subinterval (x_i, x_{i+1}).

The problem that we need to solve is

$$\begin{aligned} u_i &= 0 && i = 0, \\ \frac{-u_{i-1} + 2u_i - u_{i+1}}{h^2} &= f_i && i = 1, \dots, n-1, \\ u_i &= 0 && i = n. \end{aligned} \qquad (P)$$

Then, let us collect al the unknowns $\{u_i\}_{i=0}^n$ in a vector $\mathbf{u}$. Then, (P) is a linear system

$$A\mathbf{u} = \mathbf{f}.$$

In this exercise we will show how to use direct methods to solve linear systems, and in particular we will discuss the **LU** and **Cholesky** decompositions that you have studied in Lecture 07.

First of all, let use define n and $\{x_i\}_{i=0}^n$.

[32]:
```
%matplotlib inline
from numpy import *
from matplotlib.pyplot import *

n = 33
h = 1./(n-1)

x=linspace(0,1,n)
```

Let us define the left-hand side matrix A.

[33]:
```
a = -ones((n-1,)) # Offdiagonal entries
b = 2*ones((n,)) # Diagonal entries
A = (diag(a, -1) + diag(b, 0) + diag(a, +1))
A /= h**2

print(A)
print(linalg.cond(A))
```

```
[[ 2048. -1024.     0. …,     0.     0.     0.]
 [-1024.  2048. -1024. …,     0.     0.     0.]
 [    0. -1024.  2048. …,     0.     0.     0.]
 …,
 [    0.     0.     0. …,  2048. -1024.     0.]
 [    0.     0.     0. …, -1024.  2048. -1024.]
 [    0.     0.     0. …,     0. -1024.  2048.]]
467.842628839
```

Moreover, let us choose

$$f(x) = x(1-x)$$

so that the solution $u(x)$ can be computed analytically as

$$u(x) = u_{\text{ex}}(x) = \frac{x^4}{12} - \frac{x^3}{6} + \frac{x}{12}$$

The right hand side **f** then is easily assembled as:

[34]:
```
f = x*(1.-x)
```

We still need to impose the boundary conditions at $x = 0$ and $x = 1$, which read

$$u_i = 0 \qquad\qquad i = 0,$$

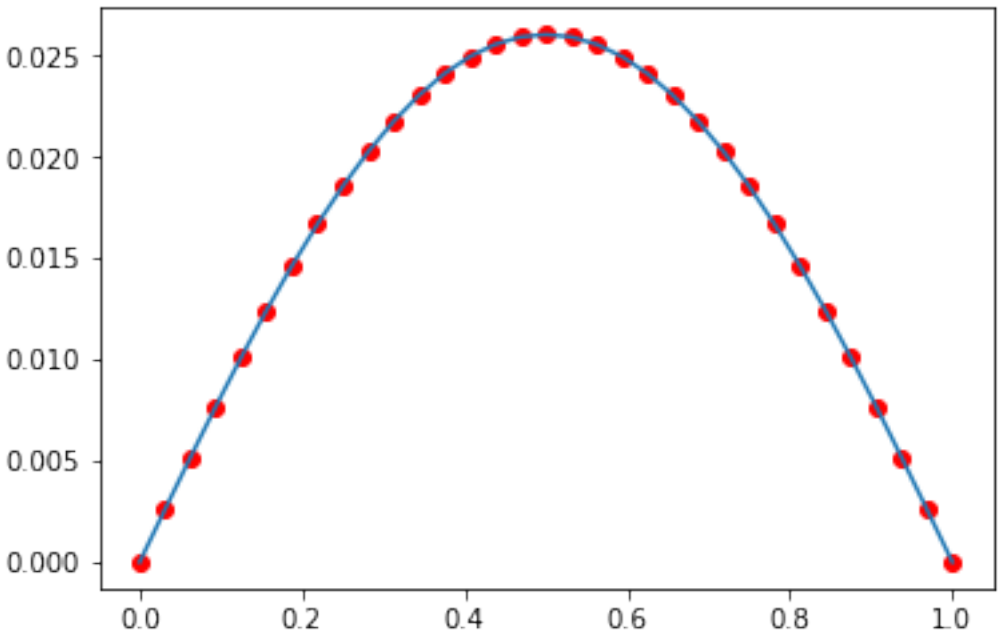
and

$$u_i = 0 \qquad\qquad i = n,$$

These conditions are associated with the first (last, respectively) row of the linear system.

Then we can solve the linear system and compare the FD approximation of u to the exact solution u_{ex}.

[35]:
```
# Change first row of the matrix A
A[0,:] = 0
A[:,0] = 0
A[0,0] = 1
f[0] = 0

# Change last row of the matrix A
A[-1,:] = 0
A[:,-1] = 0
A[-1,-1] = 1
f[-1] = 0

# Solve the linear system using numpy
A1 = A.copy()
u = linalg.solve(A1, f)
u_ex = (x**4)/12. - (x**3)/6. + x/12.

# Plot the FD and exact solution
_ = plot(x,u,'ro')
_ = plot(x,u_ex)
```



1.1 LU decomposition

We want to implement our linear solver using an **LU decomposition** (without pivoting)

$$A = LU$$

LU decomposition can be computed as in the following function.

[36]:
```
def LU(A):
    A = A.copy()
    N=len(A)
    for k in range(N-1):
        if (abs(A[k,k]) < 1e-15):
            raise RuntimeError("Null pivot")

        A[k+1:N,k] /= A[k,k]
        for j in range(k+1,N):
            A[k+1:N,j] -= A[k+1:N,k]*A[k,j]

    L=tril(A)
    for i in range(N):
        L[i,i]=1.0
    U = triu(A)
    return L, U

L, U = LU(A)
```

Once L and U have been computed, the system

$$A\mathbf{u} = \mathbf{f}$$

can be solved in **two steps**: first solve

$$L\mathbf{w} = \mathbf{f},$$

where L is a **lower triangular matrix**, and then solve

$$U\mathbf{u} = \mathbf{w}$$

where U is an **upper triangular matrix**.

These two systems can be easily solved by forward (backward, respectively) substitution.

[37]:
```
def L_solve(L,rhs):
    x = zeros_like(rhs)
    N = len(L)

    x[0] = rhs[0]/L[0,0]
    for i in range(1,N):
        x[i] = (rhs[i] - dot(L[i, 0:i], x[0:i]))/L[i,i]
```

```
    return x
```

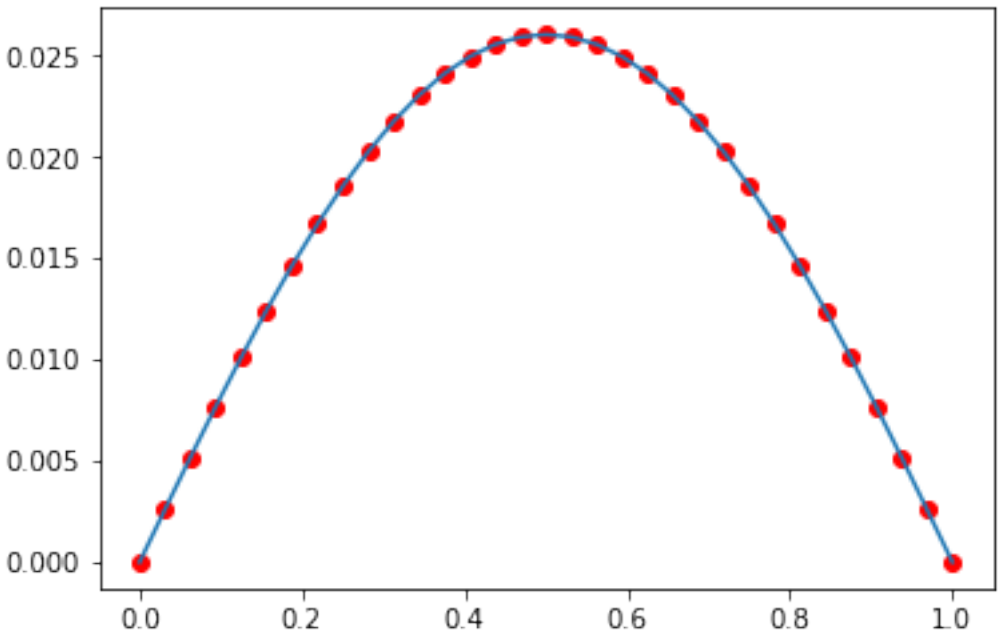
[38]:
```
def U_solve(U,rhs):
    pass # TODO
```

Now let's solve the system

$$A\mathbf{u} = \mathbf{f}$$

and compare the solution with respect to the exact solution.

[39]:
```
w = L_solve(L,f)
u = U_solve(U,w)

_ = plot(x,u,'ro')
_ = plot(x,u_ex)
```



1.1.1 try to compute the solution $u(x)$ with different forcing terms and compare with the exact solution without recomputing the LU decomposition

[40]:
```
# YOUR CODE HERE
```

1.2 Cholesky decomposition

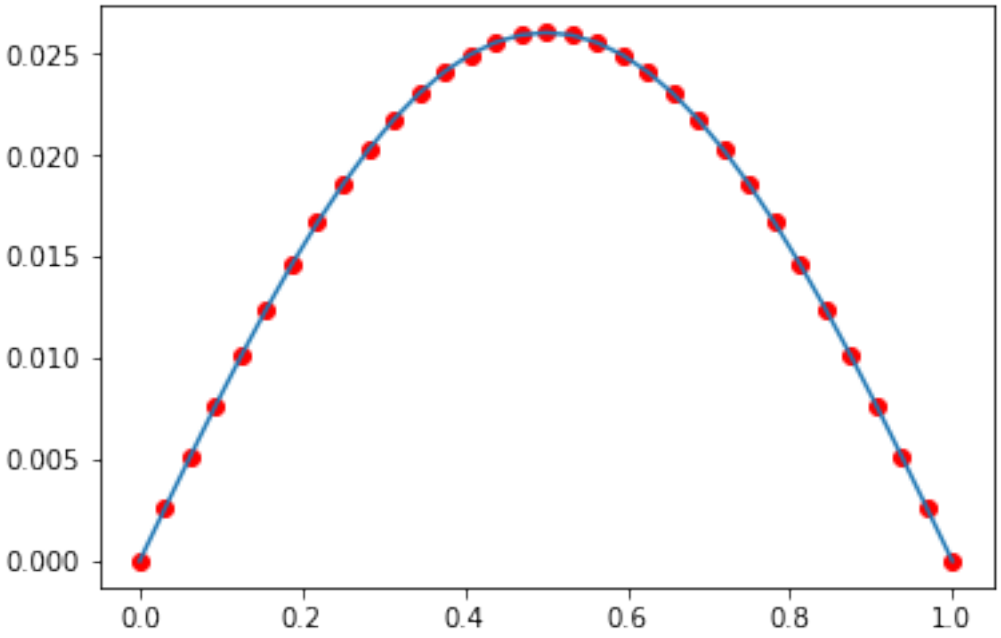
For symmetric and positive define matrices, the Cholesky decomposition may be preferred since it reduces the number of flops for computing the LU decomposition by a factor of 2.

The Cholesky decomposotion seeks an upper triangular matrix H (with all positive elements on the diagonal) such that

$$A = H^T H$$

An implementation of the Cholesky decomposition is provided in the following function. We can use it to solve the linear system by forward and backward substitution.

[41]:
```
def cholesky(A):
    A = A.copy()
    N = len(A)
    for k in range(N-1):
        A[k,k] = sqrt(A[k,k])
        A[k+1:N,k] = A[k+1:N,k]/A[k,k]

        for j in range(k+1,N):
            A[j:N,j] = A[j:N,j] - A[j:N,k]*A[j,k]

    A[-1,-1] = sqrt(A[-1,-1])
    L=tril(A)
    return L, L.transpose()


HT, H = cholesky(A)
y = L_solve(HT,f)
u = U_solve(H,y)


_ = plot(x,u,'ro')
_ = plot(x,u_ex)
```


0.025
0.020
0.015
0.010
0.005
0.000
0.0
0.2
0.4
0.6
0.8
1.0

05b_least_squares

October 16, 2019

1 Least squares

The following is an example from Lecture 06.

The result of census of the population of Switzerland between 1900 and 2010 (in thousands) is summarized in the following table:

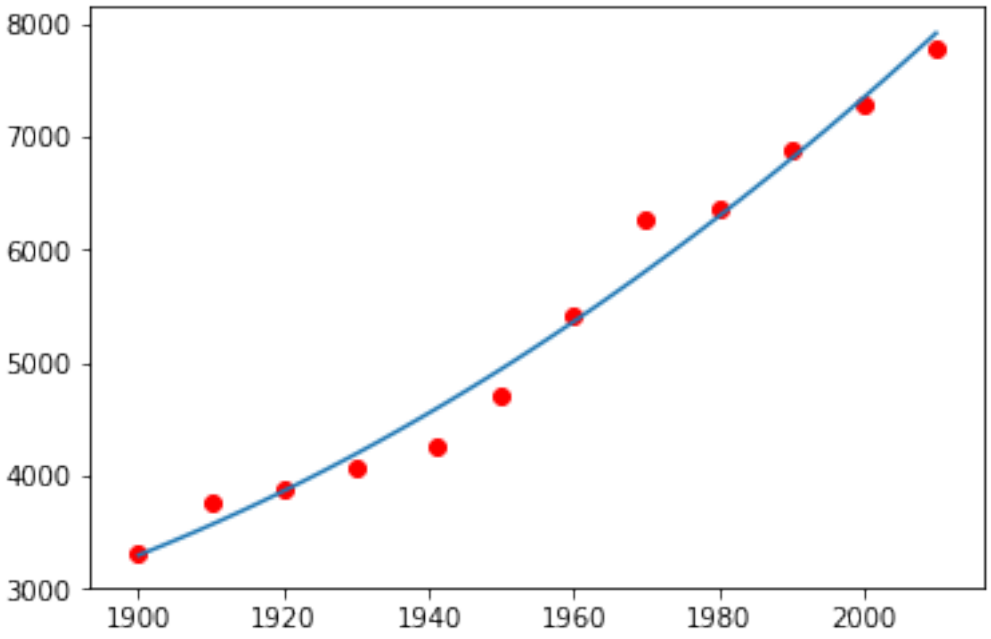
year	population
1900	3315
1910	3753
1920	3880
1930	4066
1941	4266
1950	4715
1960	5429
1970	6270
1980	6366
1990	6874
2000	7288
2010	7783

- Is it possible to estimate the number of inhabitants of Switzerland during the year when there has not been census, for example in 1945 and 1975?
- Is it possible to predict the number of inhabitants of Switzerland in 2020?

[5]:
```
%matplotlib inline
from numpy import *
from matplotlib.pyplot import *

year = array([1900, 1910, 1920, 1930, 1941, 1950, 1960, 1970, 1980, 1990, 2000,␣
 ↪2010])
population = array([3315, 3753, 3880, 4066, 4266, 4715, 5429, 6270, 6366, 6874,␣
 ↪7288, 7783])

assert len(year) is len(population)
n = len(year)
B = matrix([ones(n), year, year**2]).T
```

```
BT = B.T

y = matrix(population).T

a = linalg.solve(BT*B, BT*y)
a0 = float(a[0])
a1 = float(a[1])
a2 = float(a[2])
print(a0, a1, a2)

xx = linspace(1900,2010,100)
f = lambda x: a0 + a1*x + a2*x**2

_ = plot(xx, f(xx))
_ = scatter(year, population, color='r')



print(f(1945))
print(f(1975))
print(f(2020))
```

```
501596.69896291976 -549.8998014613501 0.1513877127640438
4745.087199790636
6051.288177001872
8521.523173396941
```



05c_linear_systems_direct_2

October 16, 2019

1 Direct methods for solving linear systems (homework)

Exercise 1. Let us consider the linear system $A\mathbf{x} = \mathbf{b}$ where

$$A = \begin{bmatrix} \epsilon & 1 & 2 \\ 1 & 3 & 1 \\ 2 & 1 & 3 \end{bmatrix}.$$

1. Find the range of values of $\epsilon \in \mathbb{R}$ such that the matrix A is symmetric and positive definite. **Suggestion**: use the *Sylvester's criterion* which states that a symmetric matrix $A \in \mathbb{R}^{n\times n}$ is positive definite if and only if all the main minors (The main minors of $A \in \mathbb{R}^{n\times n}$ are the determinants of the submatrices $A_p = (a_{i,j})_{1\leq i,j\leq p}$, $p = 1, ..., n$). of A are positive.
2. What factorization is more suitable for solving the linear system $A\mathbf{x} = \mathbf{b}$ for the case $\epsilon = 0$? Motivate the answer.
3. Compute the Cholesky factorization $A = R^T R$ for the case $\epsilon = 2$.
4. Given $\mathbf{b} = (1, 1, 1)^T$, solve the linear system by using the Cholesky factorization computed at the previous point.

Exercise 2. Let us consider the following matrix $A \in \mathbb{R}^{3\times 3}$ depending on the parameter $\epsilon \in \mathbb{R}$:

$$A = \begin{bmatrix} 1 & \epsilon & -1 \\ \epsilon & \frac{35}{3} & 1 \\ -1 & \epsilon & 2 \end{bmatrix}.$$

1. Calculate the values of the parameter $\epsilon \in \mathbb{R}$ for which the matrix A is invertible (non singular).
2. Calculate the Gauss factorization LU of the matrix A (when non singular) for a generic value of the parameter $\epsilon \in \mathbb{R}$.
3. Calculate the values of the parameter $\epsilon \in \mathbb{R}$ for which the Gauss factorization LU of the matrix A (when non singular) exists and is unique.
4. Set $\epsilon = \sqrt{\frac{35}{3}}$ and use the pivoting technique to calculate the Gauss factorization LU of the matrix A.
5. For $\epsilon = 1$, the matrix A is symmetric and positive definite. Calculate the corresponding Cholesky factorization of the matrix A, i.e. the upper triangular matrix with positive elements on the diagonal, say R, for which $A = R^T R$.

07a_ode

October 16, 2019

1 ODE

We will solve the following linear Cauchy model

$$y'(t) = \lambda y(t) \tag{1}$$
$$y(0) = 1 \tag{2}$$

whose exact solution is

$$y(t) = e^{\lambda t}$$

[1]:
```
%matplotlib inline
from numpy import *
from matplotlib.pyplot import *
import scipy.linalg
import numpy.linalg

l = -5.
t0 = 0.
tf = 10.
y0 = 1.

s = linspace(t0,tf,5000)

exact = lambda x: exp(l*x)
```

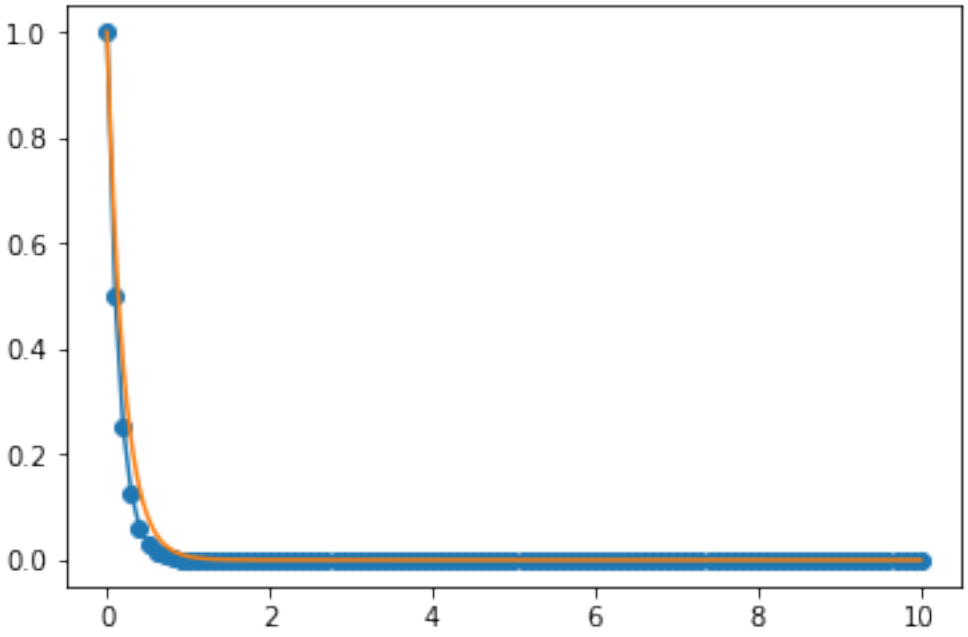
1.0.1 Forward Euler

$$\frac{y_n - y_{n-1}}{h} = f(y_{n-1}, t_{n-1})$$

[8]:
```
def fe(l,y0,t0,tf,h):
    timesteps = arange(t0,tf+1e-10, h)
    sol = zeros_like(timesteps)
    sol[0] = y0
```

```
    for i in range(1,len(sol)):
        sol[i] = sol[i-1]*(1+l*h)

    return sol, timesteps

y, t = fe(l,y0,t0,tf,0.1)

_ = plot(t,y, 'o-')
_ = plot(s,exact(s))

error = numpy.linalg.norm(exact(t) - y, 2)
print(error)
```

```
0.211605395525
```



1.0.2 Backward Euler

$$\frac{y_n - y_{n-1}}{h} = f(y_n, t_n)$$

[9]:
```
def be(l,y0,t0,tf,h):
    pass # TODO
```

1.0.3 θ-method

$$\frac{y_n - y_{n-1}}{h} = \theta\, f(y_n, t_n) + (1-\theta)\, f(y_{n-1}, t_{n-1})$$

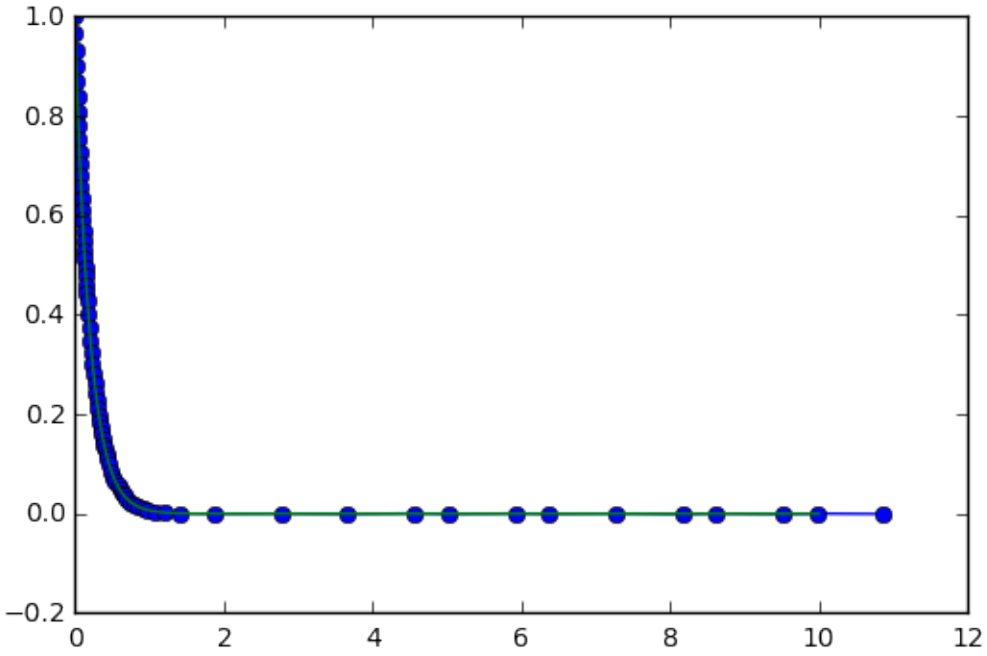
[10]:
```
def tm(theta,l,y0,t0,tf,h):
    pass # TODO
```

1.0.4 Simple adaptive time stepper

For each time step: - Compute solution with CN - Compute solution with BE - Check the difference - If the difference satisfy a given tolerance: - keep the solution of higher order - double the step size - go to the next step - Else: - half the step size and repeat the time step

[33]:
```
def adaptive(l,y0,t0,tf,h0, hmax=0.9,tol=1e-3):
    sol = []
    sol.append(y0)
    t = []
    t.append(t0)
    h = h0
    while t[-1] < tf:
        #print 'current t =', t[-1], '            h=', h
        current_sol = sol[-1]
        current_t = t[-1]
        sol_cn, _ = tm(0.5,l,current_sol,current_t, current_t + h, h)
        sol_be, _ = tm(1.,l,current_sol,current_t, current_t + h, h)

        if (abs(sol_cn[-1] - sol_be[-1]) < tol): #accept
            sol.append(sol_cn[-1])
            t.append(current_t+h)
            h *= 2.
            if h > hmax:
                h=hmax
        else:
            h /= 2.

    return sol, t

y,t = adaptive(l,y0,t0,tf,0.9)
_ = plot(t,y, 'o-')
_ = plot(s,exact(array(s)))

error = numpy.linalg.norm(exact(array(t)) - y, infty)
print error, len(y)
```

```
0.000817298421905 74
```


1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0
2
4
6
8
10
12

[ ]:

[ ]:

08_eigenvalues

October 16, 2019

1 Eigenvalue and eigenvectors calculation

$$A\mathbf{x} = \lambda\mathbf{x}$$

1.0.1 Power method (vector iteration)

- find the largest eigenvalue λ_{max}

$$\mathbf{q}_k = \frac{\mathbf{z}_{k-1}}{\|\mathbf{z}_{k-1}\|_2} \tag{1}$$

$$\mathbf{z}_k = A\mathbf{q}_k \tag{2}$$

$$\lambda_{max}^k = \mathbf{q}_k^T \mathbf{z}_k \tag{3}$$

[ ]:

[4]:
```
%matplotlib inline
from numpy import *
from matplotlib.pyplot import *
import numpy.linalg
import scipy.linalg

n = 9
h = 1./(n-1)

x=linspace(0,1,n)

a = -ones((n-1,))
b = 2*ones((n,))
A = (diag(a, -1) + diag(b, 0) + diag(a, +1))

A /= h**2

#print A
```

```
z0 = ones_like(x)


def PM(A,z0,tol=1e-5,nmax=500):
    q = z0/numpy.linalg.norm(z0,2)
    it = 0
    err = tol + 1.
    while it < nmax and err > tol:
        z = dot(A,q)
        l = dot(q.T,z)
        err = numpy.linalg.norm(z-l*q,2)
        q = z/numpy.linalg.norm(z,2)

        it += 1
    print("error =", err, "iterations =", it)
    print("lambda_max =", l)
    return l,q

l,x = PM(A,z0)

l_np, x_np = numpy.linalg.eig(A)

print("numpy")
print(l_np)
```

```
error = 8.45608648166e-06 iterations = 82
lambda_max = 249.735234086
numpy
[ 249.73523409  231.55417528  203.23651229  167.55417528  128.
    6.26476591   24.44582472   88.44582472   52.76348771]
```

1.0.2 Inverse power method

- find the eigenvalue λ **closest** to μ

$$M = A - \mu I \tag{4}$$
$$M = LU \tag{5}$$
$$\tag{6}$$
$$M\mathbf{x}_k = \mathbf{q}_{k-1} \tag{7}$$
$$\mathbf{q}_k = \frac{\mathbf{x}_k}{\|\mathbf{x}_k\|_2} \tag{8}$$
$$\mathbf{z}_k = A\mathbf{q}_k \tag{9}$$
$$\lambda^k = \mathbf{q}_k^T \mathbf{z}_k \tag{10}$$

[9]:
```
def IPM(A,x0,mu,tol=1e-5,nmax=500):
    pass # TODO

l,x = IPM(A,z0,6.)
```

```
error = 2.63101645873e-06 iterations = 3
lambda_max = 6.26476591422
```

[ ]:
```

```