Numerical Analysis – Lab 4

1 Lagrange interpolating polynomials

Implementation of Lagrange formula. Given (x_i, y_i), $i = 0, \ldots, n$ and $x_i \neq x_j$ for $i \neq j$, there exists a unique polynomial p_n of degree $\leq n$ such that $p_n(x_i) = y_i$. The Lagrange interpolation formula

$$p_n(x) = \sum_{k=0}^{n} (y_k L_{n,k}(x)) = \sum_{k=0}^{n} (y_k \prod_{\substack{i=0 \\ i \neq k}}^{n} \frac{x - x_i}{x_k - x_i})$$

may be used for evaluating p_n at x. However, a naive implementation of the formula would require $(n+1)4n + n = O(n^2)$ operations for each x. It is more attractive to reformulate

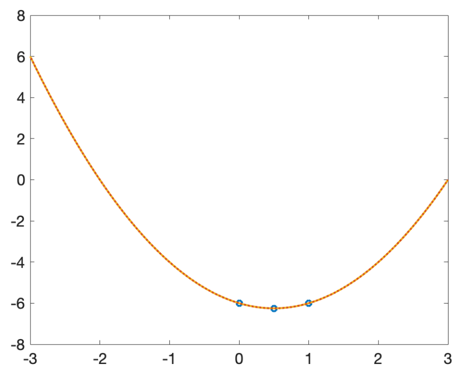
$$p_n(x) = \prod_{j=0}^{n} (x - x_j) \sum_{k=0}^{n} y_k \frac{1}{\prod_{\substack{i=0 \\ i \neq k}}^{n} (x_k - x_i)} \frac{1}{x - x_k} =: \ell(x) \sum_{k=0}^{n} \left(y_k \frac{w_k}{x - x_k} \right) \quad \text{for } x \notin \{x_i\}_{i=0}^{n}$$

because we can first compute and save $\{w_k\}_{k=0}^{n}$ then reuse them for any x. The complexity for $\{w_k\}_{k=0}^{n}$ is $(n+1)2n = O(n^2)$ but after that evaluating p_n at one x costs only $(n+1)4 = O(n)$ operations. The scheme is called the modified Lagrange formula.

```
function [yy, w] = lagrange(x,y,xx)
n = numel(x) - 1;  i = 1:n+1;
yy = zeros(size(xx));

% if xx==x then return yy=y
isx = zeros(size(xx),'logical');
for j = 1:n+1
    isxj = (x(j)==xx);
    yy(isxj) = y(j);
    isx = any([isx(:) isxj(:)],2);
end
notx = ~isx;

% first compute w for the nodes x
w = zeros(n+1,1);
for k = 1:n+1
    w(k) = 1/prod(x(k)-x(i~=k));
end

% then evaluate p at xx
for k = 1:n+1
   yy(notx)=yy(notx)+y(k)*w(k)./(xx(notx)-x(k));
end
for j = 1:n+1
    yy(notx) = yy(notx).*(xx(notx)-x(j));
end
```

Let us test our function `lagrange`. First, interpolate a quadratic function with 3 nodes:

```
>> f = @(x) x.^2 - x - 6;
>> x = 0:0.5:1;  y = f(x);
>> xx = -3:0.01:3;
>> yy = lagrange(x,y,xx);
>> plot(x,y,'o',xx,f(xx),xx,yy,':');
```



We see the interpolant coincides with the original function. Next we interpolate the analytical function e^x:

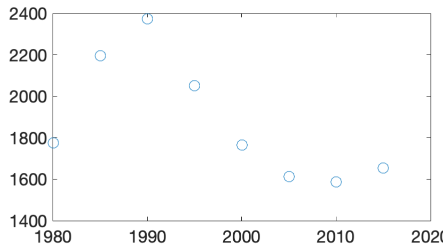
```
>> f = @(x) exp(x);
>> x = -1:0.2:1;  y = f(x);
>> xx = -2:0.01:2;
>> yy = lagrange(x,y,xx);
>> plot(x,y,'o',xx,f(xx),xx,yy,':');
```

Data interpolation and extrapolation. The birth populations of China in some years are

Year	1980	1985	1990	1995	2000	2005	2010	2015
Birth population (ten thousands)	1776	2196	2374	2052	1765	1612	1588	1655

Let p be the polynomial interpolating all the given data. Evaluate p at the years 1998 and 2020. The actual birth population was 1934 (ten thousands) in the year 1998 and 1200 (ten thousands) in the year 2020. Calling

```
>> x = 1980:5:2015;
>> y = [1776 2196 2374  2052  1765  1612  1588  1655]
>> xx = [1998 2020];
>> yy = lagrange(x,y,xx)
```



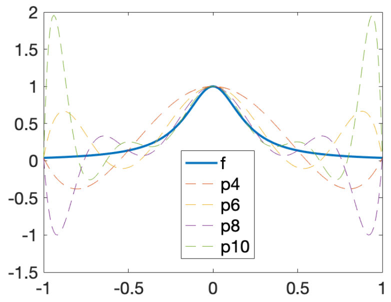
gives the number 1860 for the year 1998, and the number -270 for the year 2020. While the number 1860 looks reasonable, the negative number -270 is nonsense. The difference is that the year 1998 is between the given years, but the year 2020 is beyond the maximum of the given years. The former is called interpolation, and the latter is called extrapolation which is much more difficult. It is also very common for data interpolation that a linear interpolation is used:

```
>> x = [1995 2000]; y = [2052  1765];
>> xx = 1998; yy = lagrange(x,y,xx)
```

which gives 1880 for the year 1998.

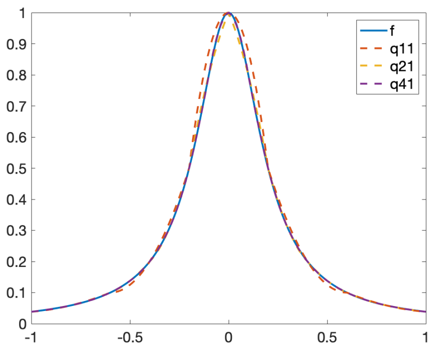
Runge phenomenon. Let $f(x) = \dfrac{1}{1+25x^2}$ and $x \in [-1,1]$. Consider the polynomial interpolation p_n of f at the nodes $x_j = -1 + \dfrac{2j}{n}$, $j = 0, 1, 2, .., n$. Plot p_4, p_6, p_8, p_{10} and f.

```
>> f = @(x) 1./(1+25*x.^2);
>> xx = -1:0.01:1;
>> plot(xx, f(xx)); hold on
>> for n = 4:2:10
     x = -1:(2/n):1;
     y = f(x);
     yy = lagrange(x,y,xx);
     plot(xx,yy,'--');
   end
```



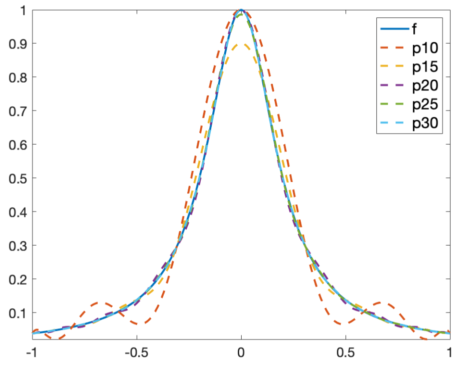
Observe that how the error near ± 1 grows as the degree of polynomial increases. One approach to overcoming the Runge phenomenon is piecewise interpolation. For example, we can divide the interval $[-1, 1]$ into 10 equally spaced subintervals and on each subinterval construct an interpolating polynomial of small degree. Plot the piecewise quadratic function for approximation of f on $[-1, 1]$ with 11, 21 and 41 equally spaced nodes. As the length of each subinterval decreases, the piecewise interpolant approximates f more accurately. Optional: indeed, one can show that $\max_{x\in[-1,1]} |q_n(x) - f(x)| \xrightarrow{n} 0$.

```
>> for m = 0:2
     nn = 10*2^m;
     x = -1:(2/nn):1;
     y = f(x);
     for i=1:2:nn-1
         sub = xx>=x(i) & xx<=x(i+2);
         yy(sub)=lagrange(x(i:i+2),y(i:i+2),xx(sub));
     end
     plot(xx,yy,'--');
   end
```



Another approach to overcoming the Runge phenomenon is to distribute the nodes non-evenly and denser near the interval ends. For example, use the Chebyshev nodes $x_k = \cos\frac{k\pi}{n}$, $k = 0, \ldots, n$.

```
>> for n = 10:5:30
     x = cos((0:n)*pi/n);
     y = f(x);
     yy = lagrange(x,y,xx);
     plot(xx,yy,'--');
   end
```



From the figure on the right, we can find that as number of nodes increase the interpolant approximates f more accurately. Optional: indeed, one can show that

$$\max_{x\in[-1,1]} |p_n(x) - f(x)| \overset{n}{\to} 0.$$

2 Topics in interpolation (optional)

Bernstein polynomials. Sergei Bernstein (1880-1968) introduced his polynomial basis

$$B_{k,n}(x) = \binom{n}{k} \frac{1}{(b-a)^n}(x-a)^k(b-x)^{n-k}$$

for approximation of $f \in C[a,b]$. The polynomial approximation found by Bernstein is

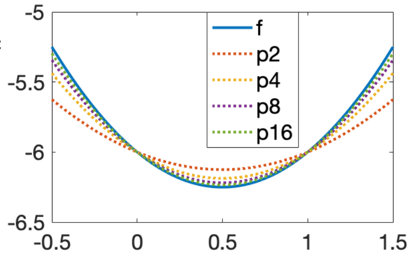
$$f(x) \approx p_n(x) = \sum_{k=0}^{n} f\left(a + \frac{k}{n}(b-a)\right) B_{k,n}(x),$$

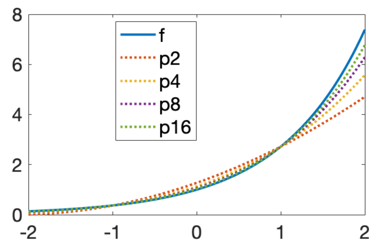
which was shown by Bernstein to converge: $\max_{[a,b]} |p_n(x) - f(x)| \overset{n}{\to} 0$. Let us program it:

```
function [yy, w] = bernstein(a,b,y,xx)
n = numel(y) - 1;
yy = zeros(size(xx));
for k = 0:n
    yy = yy + y(k+1)*nchoosek(n,k)*(xx-a).^k.*(b-xx).^(n-k)/(b-a)^n;
end
```

Let us test our function `bernstein`. First, approximate a quadratic function with $n = 2, 4, 8, 16$ and $[a,b] = [0,1]$:

```
>> f = @(x) x.^2 - x - 6;
>> xx = -0.5:0.01:1.5; plot(xx,f(xx),'linewidth',2);
>> hold on
>> for m = 1:4
      n = 2^m;
      x = linspace(0,1,n+1);  y = f(x);
      yy = bernstein(0,1,y,xx);
      plot(xx,yy,':','linewidth',2);
   end
```



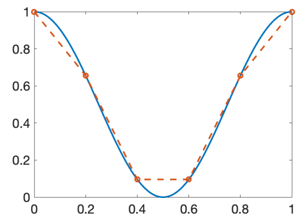


Next we approximate the analytical function e^x with $[a,b] = [-1,1]$. You can adapt the above commands to this case. Observe how p_n converges uniformly to f on $[a,b]$ and even beyond $[a,b]$.

The Bernstein polynomial approximation is not an interpolating polynomial. It equals f at the two ends of $[a,b]$ but not elsewhere. But it guarantees the uniform convergence as long as $f \in C[a,b]$ and uses equally spaced nodes– free of the Runge phenomenon!

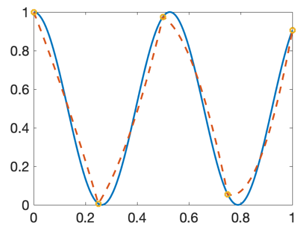
Splines. When one uses the piecewise interpolation, in particular, the piecewise linear interpolation, one observes the nonsmoothness at the nodes joining two pieces (subintervals). For example,

```
>> x = 0:0.2:1; f = @(x) cos(pi*x).^2;
>> xx = 0:0.002:1;
>> plot(xx,f(xx),x,f(x),'o--','linewidth',2);
```



(We did not actually compute the piecewise linear interpolant at `xx` for the plot. Because MATLAB automatically plots the points with straighlines in between.) The nonsmooth transition between adjacent pieces occurs also for higher degree interpolants.

```
>> f = @(x) cos(1.9*pi*x).^2;
>> x = 0:0.125:1; y = f(x);
>> for i=1:2:7
      sub = xx>=x(i) & xx<=x(i+2);
      yy(sub)=lagrange(x(i:i+2),y(i:i+2),xx(sub));
   end
>> plot(xx,f(xx),xx,yy,'--',x(1:2:end),y(1:2:end),'o');
```



The sought of smooth graphs motivates use of *splines* which do piecewise approximation but require smooth transition at the joining nodes. The most popular splines are the cubic splines for which each subinterval $[x_i, x_{i+1}]$ contains only the two nodes $\{x_i, x_{i+1}\}$ and a cubic approximation p_i is defined on $[x_i, x_{i+1}]$ such that

(i) $p_i(x_i) = f(x_i)$, $p_i(x_{i+1}) = f(x_{i+1})$,

(ii) $p_i'(x_i) = p_{i-1}'(x_i)$, $p_i''(x_i) = p_{i-1}''(x_i)$, $p_i'(x_{i+1}) = p_{i+1}'(x_{i+1})$, $p_i''(x_{i+1}) = p_{i+1}''(x_{i+1})$.

Suppose the set of nodes (or called knots) is $a = x_0 < x_1 < \ldots < x_{n-1} < x_n = b$. Let

$$p_i(x) = f(x_i) + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3.$$

At the joining nodes $\{x_1, \ldots, x_{n-1}\}$, we get from the continuity conditions (i)-(ii) that

$$\begin{cases} f(x_1) = f(x_0) + b_0(x_1 - x_0) + c_0(x_1 - x_0)^2 + d_0(x_1 - x_0)^3 \\ \qquad \vdots \\ f(x_n) = f(x_{n-1}) + b_{n-1}(x_n - x_{n-1}) + c_{n-1}(x_n - x_{n-1})^2 + d_{n-1}(x_n - x_{n-1})^3 \end{cases}$$
$$\begin{cases} 0 = b_0 + 2c_0(x_1 - x_0) + 3d_0(x_1 - x_0)^2 - b_1 \\ \ \vdots \\ 0 = b_{n-2} + 2c_{n-2}(x_{n-1} - x_{n-2}) + 3d_{n-2}(x_{n-1} - x_{n-2})^2 - b_{n-1} \end{cases}$$
$$\begin{cases} 0 = 2c_0 + 6d_0(x_1 - x_0) - 2c_1 \\ \ \vdots \\ 0 = 2c_{n-2} + 6d_{n-2}(x_{n-1} - x_{n-2}) - 2c_{n-1} \\ 0 = 2c_{n-1} + 6d_{n-1}(x_n - x_{n-1}) - 2c_n \ \text{(definition of } c_n) \end{cases}$$

From the last subsystem, we can solve out d_i's in terms of c_i's:

$$d_i = \frac{c_{i+1} - c_i}{3(x_{i+1} - x_i)} \quad \text{for } i = 0, \ldots, n-1.$$

Substituting this into the first subsystem, we can solve out b_i's in terms of c_i's:

$$\begin{aligned} b_i &= \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} - c_i(x_{i+1} - x_i) - d_i(x_{i+1} - x_i)^2 \\ &= \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} - \frac{x_{i+1} - x_i}{3}(2c_i + c_{i+1}), \quad \text{for } i = 0, \ldots, n-1. \end{aligned}$$

Substituting d_i and b_i into the second subsystem, we will find

$$3\left(\frac{f(x_{i+2})-f(x_{i+1})}{x_{i+2}-x_{i+1}}-\frac{f(x_{i+1})-f(x_i)}{x_{i+1}-x_i}\right)=(x_{i+1}-x_i)c_i+2(x_{i+2}-x_i)c_{i+1}+(x_{i+2}-x_{i+1})c_{i+2}$$

for $i=0,\ldots,n-2$. Compared to the number of unknowns $\{c_0,\ldots,c_n\}$, we still need two more conditions. To have a unique solution for the spline interpolation, it is common to specify two extra conditions: one at x_0 and the other at x_n.

(1) Natural boundary condition: $p_0''(x_0)=p_{n-1}''(x_n)=0$. This gives $c_0=c_n=0$. Note that, however, the natural spline converges to f only of the order $O(h^2)$ for $h=\max_i|x_i-x_{i-1}|$.

(2) Not-a-knot condition of de Boor: $p_0(x)\equiv p_1(x)$ and $p_{n-2}(x)\equiv p_{n-1}(x)$. Or equivalently, $p_0'''(x_1)=p_1'''(x_1)$ and $p_{n-1}'''(x_{n-1})=p_{n-2}'''(x_{n-1})$. This gives $d_0=d_1$ and $d_{n-2}=d_{n-1}$, which implies

$$\begin{aligned}0&=(x_2-x_1)c_0-(x_2-x_0)c_1+(x_1-x_0)c_2,\\0&=(x_n-x_{n-1})c_{n-2}-(x_n-x_{n-2})c_{n-1}+(x_{n-1}-x_{n-2})c_n.\end{aligned}$$

The resulting spline converges to f of the order $O(h^4)$.

(3) Periodic boundary condition: if f is periodic with the period x_n-x_0, we require $p_0'(x_0)=p_{n-1}'(x_n)$ and $p_0''(x_0)=p_{n-1}''(x_n)$. This gives

$$\begin{aligned}0&=b_{n-1}+2c_{n-1}(x_n-x_{n-1})+3d_{n-1}(x_n-x_{n-1})^2-b_0,\\0&=c_0-c_n.\end{aligned}$$

Substituting b_{n-1}, b_0, d_{n-1} in terms of c_i's and $c_n=c_0$ into the first equation, we obtain

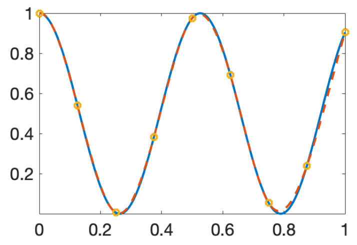
$$\begin{aligned}3\left(\frac{f(x_1)-f(x_0)}{x_1-x_0}-\frac{f(x_n)-f(x_{n-1})}{x_n-x_{n-1}}\right)&=(x_n-x_{n-1})c_{n-1}+2(x_n-x_{n-1}+x_1-x_0)c_0+(x_1-x_0)c_1,\\0&=c_n-c_0.\end{aligned}$$

We can implement the following function for the spline interpolations.

```
function [yy, b, c, d] = cubicspline(x,y,xx,type)
% type: 1 for natural, 2 for de Boor, 3 for periodic
n = numel(x)-1; dx = diff(x(:)); dy = diff(y(:)); slope = dy./dx;
left=[dx(1:end-1);0;0]; centre=[1;2*(dx(1:end-1)+dx(2:end));1]; right=[0;0;dx(2:end)];
A=spdiags([left centre right],-1:1,n+1,n+1); rhs=[0;3*(slope(2:end)-slope(1:end-1));0];
if 1==type
    % already done
elseif 2==type
    A(1,1)=dx(2); A(1,2)=-(dx(1)+dx(2)); A(1,3)=dx(1);
    A(end,end)=dx(end-1); A(end,end-1)=-(x(end)-x(end-2)); A(end,end-2)=dx(end);
elseif 3==type
    if abs(y(end)-y(1))>5e-16
        disp('cubicspline: WARNING: non-periodic y for type periodic');
    end
    A(1,1)=2*(dx(end)+dx(1)); A(1,2)=dx(1); A(1,end-1)=dx(end);
    rhs(1)=3*(slope(1)-slope(end));  A(end,1)=-1;
end
c=A\rhs; b=slope-dx/3.*(2*c(1:end-1)+c(2:end)); d=diff(c)./dx/3; yy=zeros(size(xx));
for i=1:n
    sub=xx>=x(i) & xx<=x(i+1);  fac=xx(sub)-x(i);
    yy(sub)=y(i)+(b(i)+(c(i)+d(i)*fac).*fac).*fac;
end
```

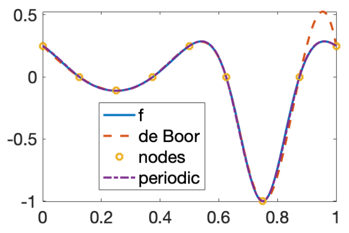
Let us try the spline interpolation and see if the spline is smooth.

```
>> f = @(x) cos(1.9*pi*x).^2;
>> x = 0:0.125:1; y = f(x); xx = 0:0.002:1;
>> yy = cubicspline(x,y,xx,2);
>> plot(xx,f(xx),xx,yy,'--',x,y,'o');
```



Compared to our previous experiment with the piecewise quadratic Lagrange interpolation, the cubic spline interpolant is smooth everywhere and also looks more accurate using the same number of nodes. Next we approximate a periodic function for one period.

```
>> f = @(x) cos(4*pi*x)./(2+sin(2*pi*x)).^2;
>> x = 0:0.125:1; y = f(x); xx = 0:0.002:1;
>> yy2 = cubicspline(x,y,xx,2); yy3 = cubicspline(x,y,xx,3);
>> plot(xx,f(xx),xx,yy2,'--',x,y,'o',xx,yy3,'-.');
```



We see that the de Boor spline does not capture the periodicity $f'(0) = f'(1)$, while the periodic spline does and gives much more accurate result.

Trigonometric interpolation. Suppose f is a 1-period function. We seek a trigonometric sum

$$p_n(x) = \sum_{k=-n+\lfloor n/2\rfloor+1}^{\lfloor n/2\rfloor} c_k e^{ik2\pi x} \qquad (\lfloor x\rfloor \text{ is the floor of a number } x \text{ i.e. the biggest integer} \le x)$$

such that, for $x_j = j/n$, $j = 0, \dots, n-1$, the interpolation condition holds

$$f(x_j) = p_n(x_j) = \sum_{k=-n+\lfloor n/2\rfloor+1}^{\lfloor n/2\rfloor} c_k e^{ik2\pi x_j}.$$

Let $\mathbb{C}^n \ni \boldsymbol{\phi}_k = (e^{ik2\pi x_j})_{j=0}^{n-1}$, $k = -n + \lfloor n/2\rfloor + 1, \dots, 0, \dots, \lfloor n/2\rfloor$. It can be verified that

$$\boldsymbol{\phi}_k \cdot \boldsymbol{\phi}_m = \begin{cases} n & \text{if } k = m \\ 0 & \text{otherwise} \end{cases}$$

where the inner product for two vectors $\mathbf{u}, \mathbf{v}$ of complex numbers is defined as $\mathbf{u} \cdot \mathbf{v} := \sum_j u_j \bar{v}_j$. Let $\mathbf{y} = (f(x_j))_{j=0}^{n-1}$. Then, taking inner product with $\boldsymbol{\phi}_m$ on both sides of

$$\mathbf{y} = \sum_{k=-n+\lfloor n/2\rfloor+1}^{\lfloor n/2\rfloor} c_k \boldsymbol{\phi}_k$$

leads to

$$c_m = \frac{1}{n}\mathbf{y} \cdot \boldsymbol{\phi}_m = \frac{1}{n}\sum_{j=0}^{n-1} f(x_j) e^{-im2\pi x_j}.$$

If we admit m to be any integer in the above equality for c_m, then we find

$$c_m = c_{m+n} \quad \text{because } e^{-i(m+n)2\pi x_j} = e^{-i(m+n)2\pi j/n} = e^{-in2\pi j/n} = e^{-im2\pi x_j}.$$

The relation between $\mathbf{y}$ and $\mathbf{c} = (c_k)_{k=0}^{n-1}$ describes the discrete Fourier transform and its inverse. There are different conventions for the definitions depending on where to put the constant $\frac{1}{n}$. According to the documentation of MATLAB, `Y = fft(X)` and `X = ifft(Y)` implement the discrete Fourier transform and discrete inverse Fourier transform, respectively. For `X` and `Y` of length n, these transforms are defined as follows:

$$Y(k) = \sum_{j=1}^{n} X(j) e^{-i(k-1)2\pi(j-1)/n}, \quad k = 1, \dots, n,$$

$$X(j) = \frac{1}{n}\sum_{k=1}^{n} Y(k) e^{i(k-1)2\pi(j-1)/n}, \quad j = 1, \dots, n.$$

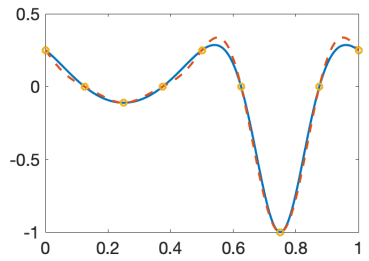
So in our case $\mathbf{c} = \frac{1}{n}$`fft`$(\mathbf{y})$ and $\mathbf{y}$ =`ifft`$(n\mathbf{c})$. Once we have $\mathbf{c}$, we can evaluate the trigonometric interpolant p_n at any x by calculating the sum. Note that we derived the above formulae only for the interval $[0,1]$. For f a general interval $[a,b]$, we can define $\tilde{f}(t) = f((1-t)a + tb)$ on $[0,1]$ and find the trigonometric interpolant $\tilde{p}_n(t)$ for $\tilde{f}$, then define

$$p_n(x) = \tilde{p}_n\left(\frac{x-a}{b-a}\right).$$

```
function [yy, c] = triginterps(a,b,y,xx)
% y corresponds to function values at
% equally spaced nodes on [a, b).
n=numel(y); c=fft(y(:))/n; t=(xx-a)/(b-a);
z=exp(1i*2*pi*t); nh=floor(n/2);
yy=c(nh+1)*ones(size(xx));
for k=nh:-1:1
    yy = c(k) + yy.*z;
end
yyl=c(nh+2)*ones(size(xx)); z=conj(z);
for k=nh+3:n
    yyl = c(k) + yyl.*z;
end
yy = yy + yyl.*z;
if isreal(y)
    yy = real(yy);
end
```

We interpolate the periodic function by the trigonometric polynomial:

```
>> f = @(x) cos(4*pi*x)./(2+sin(2*pi*x)).^2;
>> x = 0:0.125:1; y = f(x); xx = 0:0.002:1;
>> yy = triginterps(0,1,y(1:end-1),xx);
>> plot(xx,f(xx),xx,yy,'--',x,y,'o');
```



It looks a little less accurate than the periodic spline shown previously.

If we want to evaluate the trigonometric interpolant at $N >> n$ different x's (but equally spaced), we would better use the discrete inverse Fourier transform because it costs $O(N \log N)$ operations– possibly faster than doing the summation of the n terms for N times. To this end, we view our interpolant

$$p_n(x) = \sum_{k=-n+\lfloor n/2\rfloor+1}^{\lfloor n/2\rfloor} c_k \mathrm{e}^{\mathrm{i}k2\pi x}$$

as

$$p_N(x) = \sum_{k=-N+\lfloor N/2\rfloor+1}^{\lfloor N/2\rfloor} d_k \mathrm{e}^{\mathrm{i}k2\pi x},$$

with

$$d_k = \begin{cases} c_k & \text{for } k \in \{-n+\lfloor n/2\rfloor+1,\ldots,\lfloor n/2\rfloor\} \\ 0 & \text{for } k \in \{-N+\lfloor N/2\rfloor+1,\ldots,-n+\lfloor n/2\rfloor\} \cup \{\lfloor n/2\rfloor+1,\ldots,\lfloor N/2\rfloor\}. \end{cases}$$

Moreover, extend the sequence periodically $d_k = d_{k+N}$. Then evaluating p_N at $\{X_j = j/N\}_{j=0}^{N-1}$ gives

$$Y_j = p_N(X_j) = \sum_{k=-N+\lfloor N/2\rfloor+1}^{\lfloor N/2\rfloor} d_k \mathrm{e}^{\mathrm{i}k2\pi X_j} = \sum_{k=0}^{N-1} d_k \mathrm{e}^{\mathrm{i}k2\pi X_j}.$$

Let $\mathbf{Y} = (Y_j)_{j=0}^{N-1}$, $\mathbf{d} = (d_k)_{k=0}^{N-1}$. So in MATLAB it corresponds to **Y**=`N*ifft(`**d**`)`. We implement this as follows.

```
function [Y,X] = triginterpf(a,b,y,N)
% evaluate at X = a + (b-a)*j/N, j=0,..,N
n=numel(y); c=fft(y(:))/n; X=linspace(a,b,N+1); nh=floor(n/2);
d=zeros(N,1); d(1:nh+1)=c(1:nh+1); d(N-n+nh+2:N)=c(nh+2:n);
Y=N*ifft(d); if isreal(y), Y=real(Y); end; Y(end+1)=Y(1);
```

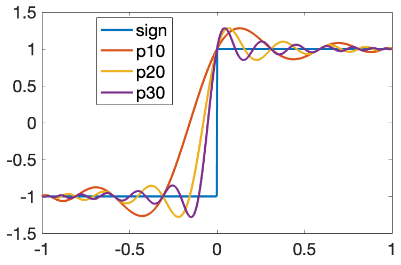
Gibbs phenomenon.

When we approximate a function with a discontinuity with smooth functions (e.g. polynomials or a Fourier series), the approximations oscillate and overshoot near the discontinuity. That phenomenon is named after Gibbs. Let

$$f(x) = \operatorname{sign}(x) = \begin{cases} 1 & \text{for } x > 0 \\ 0 & \text{for } x = 0 \\ -1 & \text{for } x < 0 \end{cases}.$$

We try interpolating f in the Chebyshev nodes on $[-1, 1]$:

```
>> xx = -1:0.001:1; plot(xx,sign(xx)); hold on;
>> for n = 10:10:30
       x = cos((0:n)*pi/n); y = sign(x);
       yy = lagrange(x,y,xx); plot(xx,yy);
   end
```

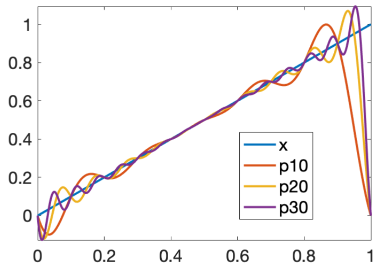


From the figure, we can find that the interpolating polynomials have overshoots near the discontinuity and they also become more oscillating as n increases.

In fact, the positive overshoots will tend to a constant height $1.2822\ldots$ as $n \to \infty$.

The discontinuity can also come from the periodic assumption on the function. For example, if we approximate $f(x) = x$ on $[0, 1]$ with the trigonometric interpolant of period 1, then the periodicity of f is broken at $x = 0$ i.e. $f(0) \neq f(1)$, which gives the discontinuities of the periodic extension of f. That has an impact on the trigonometric interpolation:

```
>> xx = 0:0.001:1; plot(xx,xx); hold on;
>> for n = 10:10:30
       x = linspace(0,1,n+1); y = x;
       yy = triginterps(0,1,y(1:end-1),xx); plot(xx,yy)
   end
```



From the figure, we can find that the interpolating trigonometric polynomials have overshoots near the periodic discontinuities of f and they also become more oscillating as n increases.

The issue of Gibbs phenomenon may be resolved in some way. For example, by detecting the discontinuity then making piecewise approximation, or using some postprocessing filters. But that is beyond the scope of this module.