Numerical Analysis – Lab 5

1 Neville's method (optional)

Tabulated integral. Let

$$f(x) = \int_0^x \mathrm{e}^{\sin t} \, \mathrm{d}t.$$

If we know the tabulated values

x	0.4	0.5	0.6	0.7	0.8	0.9
$f(x)$	0.4904	0.6449	0.8136	0.9967	1.1944	1.4063

what is the value $f(0.66)$? We can compute a sequence of interpolations using more and more data around 0.66 by Neville's method:

```
function [yy, T] = neville(x,y,xx)
% works only for a scalar xx
n = numel(x); T = zeros(n);
for i = 1:n
    T(i,1) = y(i);
    for j=1:i-1
        T(i,j+1)=((x(i)-xx)*T(i-1,j)+(xx-x(i-j))*T(i,j))/(x(i)-x(i-j));
    end
end
yy = T(n,n);
```

saved in the file `neville.m` and invoked with

```
>> x = 0.4:0.1:0.9;
>> y = [0.4904  0.6449  0.8136  0.9967  1.1944  1.4063];
>> xx = 0.66; [~,I] = sort(abs(x-xx)); x = x(I); y = y(I);
>> [yy,Q] = neville(x,y,xx);
```

```
    x      y
-----------------------------------
   0.7   0.9967   0          0          0          0          0
   0.6   0.8136   0.92346    0          0          0          0
   0.8   1.1944   0.92784    0.92171    0          0          0
   0.5   0.6449   0.93797    0.92176    0.92172    0          0
   0.9   1.4063   0.94946    0.92188    0.92179    0.92171    0
   0.4   0.4904   0.96667    0.92193    0.92189    0.92175    0.92171
```

Along the diagonal, we can see that the approximations of $f(0.66)$ seem converging to 0.92171.

Extrapolation. When x is outside the range of the nodes $\{x_i\}_{i=0}^n$, the approximation $p_n(x) \approx f(x)$ is called an extrapolation. Extrapolation is often used to compute limits

$$\lim_{t \to x} f(t).$$

Let $h = t - x$ and $T(h) = f(x + h)$. Then the limit becomes

$$\lim_{t \to x} f(t) = \lim_{h \to 0} T(h).$$

Acoordingly, the approximation becomes $P_n(h) := p_n(x + h)$ and we expect

$$\lim_{n \to \infty} P_n(0) = \lim_{h \to 0} T(h),$$

which is the case if $T(h)$ admits the asymptotic expansion

$$T(h) = a_0 + a_1 h + \ldots + a_k h^k + O(h^{k+1}).$$

The data for constructing $P_n(h)$ is on a sequence $\{h_i\}_{i=0}^n$ with $h_{i+1} < ch_i$ for some constant $c \in (0,1)$. Recall Neville's formula:

$$Q_{ij}(h) = \frac{(h - h_{i-j})Q_{i,j-1}(h) - (h - h_i)Q_{i-1,j-1}(x)}{h_i - h_{i-j}}.$$

Now $h = 0$ leads to

$$Q_{ij}(0) = \frac{h_i Q_{i-1,j-1} - h_{i-j}Q_{i,j-1}}{h_i - h_{i-j}}.$$

We can apply it to compute the derivative (see also Lab 1):

$$T(h) = \frac{f(x+h) - f(x)}{h} \approx f'(x).$$

```
>> f = @sin; h = 10.^(-2:-1:-8);
>> y = (f(1+h)-f(1))./h;
>> [yy, Q] = neville(h,y,0);
>> format long; [y(:) diag(Q)]
```

which showed, the data $\{y_i\} = \{T(h_i)\}$ and the extrapolated values $\{Q_{i,i}(0)\}$, as

```
0.536085981011869    0.536085981011869
0.539881480360327    0.540303202510156
0.540260231418621    0.540302305903476
0.540298098505865    0.540302305869974
0.540301885121330    0.540302305855319
0.540302264040449    0.540302306145786
0.540302302898255    0.540302307227831
```

while the exact value is $\cos 1$ =0.540302305868140...

The extrapolation $\{Q_{i,i}(0)\}$ is still affected by the round-off errors and reaches the best accuracy with h down to 10^{-5}, which has 3 more correct digits than the best result of the original sequence $\{T(h_i)\}$ with h down to 10^{-8}. But without knowing the exact value, how can we determine which h_i gives the best result for $\{Q_{i,i}(0)\}$ or $\{T(h_i)\}$? It seems intuitive that the best result is at the end of the stage with the frozen digits extending from left to the right, after which the digits start to unfreeze from the right to the left– a signal that the growing round-off errors come to dominate. Can one justify the intuition?

Inverse interpolation. The polynomial interpolation can also be used for finding the zero of a function: $f(x) = 0$. Suppose $f(p) = 0$, $p \in (a,b)$ and $f'(x) \neq 0$ on (a,b). Let $y_i = f(x_i)$, $i = 0,1,\ldots,n$ be given. We consider the inverse function $x = f^{-1}(y)$ with $x \in [a,b]$, for which $x_i = f^{-1}(y_i)$, $i = 0,1,\ldots,n$ are given data, and $p = f^{-1}(0)$ is to be predicted. So we can find an interpolating polynomial P_n for f^{-1} and evaluate $P_n(0) \approx p$, which is called the inverse interpolation.

Let us try the above idea on the Lambert W function $x = W(t)$ which is defined implicitly by the equation

$$t = x\mathrm{e}^x.$$

For example, $p = W(1)$ is the solution of

$$x\mathrm{e}^x - 1 = 0.$$

Let $y = f(x) = x\mathrm{e}^x - 1$. We want to approximate the inverse function $x = f^{-1}(y)$ by an interpolating polynomial $P_n(x)$ for $x \in [0,1]$ and evaluating $P_n(0) \approx p$:

```
>> f= @(x) x.*exp(x)-1;
>> n = 10; x = linspace(0,1,n+1); y=f(x);
>> [xx,Q] = neville(y,x,0);
>> [diag(Q); lambertw(1)]
```

Since Neville's method generates a sequence of approximations $\{X_i := Q_{i,i}\}$ for which $Y_i := f(X_i)$ is expected to converge to zero, we can extrapolate (Y_i, X_i) to approximate $\lim_{y\to 0} f^{-1}(y)$:

```
>> X = diag(Q); X = X(end-5:end);
>> [XX, QQ] = neville(f(X),X,0);
>> [diag(QQ); lambertw(1)]

   0.572418578858086
   0.567144573875986
   0.567143290503791
   0.567143290409786
   0.567143290409784
   0.567143290409784
   0.567143290409784
```

We see that the results converge quickly to that from the MATLAB built-in function `lambertw`.

2 Newton's divided differences.

Implementation of Newton's interpolation formula. While the recursive formula of Neville's method is convenient for generating a sequence of approximations of $f(x)$ at one point x, it costs $O(n^2)$ operations and $O(n)$ memory for one x, and needs to start over for a different value of x. Newton's divided differences compute recursively the weights a_i, $i = 0, 1, \ldots, n$ for the sequence of approximations $p_k(x) = a_0 + a_1(x - x_1) + a_k(x - x_0)\cdots(x - x_{k-1})$ in the nodes $\{x_0, \ldots, x_k\}$, for $k = 1, 2, \ldots, n$. Indeed, $a_i = f[x_0, \ldots, x_i]$ is the divided differences. To compute all of $\{a_i\}_{i=0}^n$, $O(n^2)$ operations and $O(n)$ will be needed. But once ready, the same $\{a_i\}_{i=0}^n$ can be reused for evaluation of $p_k(x)$ at any x by Horner's rule (invented earlier by Qin Jiushao):

$$p_k(x) = a_0 + (x - x_0)(a_1 + (x - x_1)(a_2 + \ldots (x - x_{k-2})(a_{k-1} + (x - x_{k-1})a_k)\ldots)),$$

which costs only $O(n)$ operations for each x. This gives the following implementation:

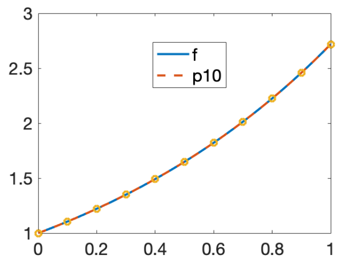
```
function A = divdif(x,y)
% A is the lower triangular matrix of divided differences including the
% zeroth divided differences i.e. the function values y
n = numel(x); x = x(:);
A = zeros(n,n);
A(:,1) = y(:);
for k = 2:n
    A(k:end,k) = diff(A(k-1:end,k-1))./(x(k:end)-x(1:end-k+1));
end
```

saved in the file `divdif.m`, and

```
function yy = newtoninterp(x,a,xx)
% evaluate the Newton polynomial with the coefficient a(0), ...
% associated with the nodes x
n = numel(x)-1;
if n~=numel(a)-1
    error('newtoninterp: x and a are of different length!');
end
yy = a(end)*ones(size(xx));
for k=n:-1:1
    yy = a(k)+(xx-x(k)).*yy;
end
```

saved in the file `newtoninterp.m`. Let us try it for $f(x) = \mathrm{e}^x$ on $[0, 1]$:

```
>> f = @exp; n = 10;
>> x = linspace(0,1,n+1); y = f(x);
>> A = divdif(x,y);
>> xx = 0:0.001:1;
>> yy = newtoninterp(x,diag(A),xx);
>> plot(xx,f(xx),xx,yy,'--',x,y,'o')
```



If we want all the approximations in the sequence $p_k(x)$ for $k = 0, \ldots, n$ then we need to use the recurrent relations $q_k(x) = q_{k-1}(x)(x - x_{k-1})$, $p_k(x) = p_{k-1}(x) + a_k q_k(x)$ with $q_0(x) = 1$, $p_0(x) = a_0$, which leads to a different implementation from `newtoninterp.m`:

```
function yyseq = newtoninterpseq(x,a,xx)
n = numel(x)-1;
if n~=numel(a)-1
    error('newtoninterpseq: x and a are of different length!');
end
yyseq = zeros(numel(xx),n+1);
yyseq(:,1) = a(1);
q = ones(numel(xx),1);
xx = xx(:);
for k = 1:n
    q = q.*(xx-x(k));
    yyseq(:,k+1) = yyseq(:,k) + a(k+1)*q;
end
```

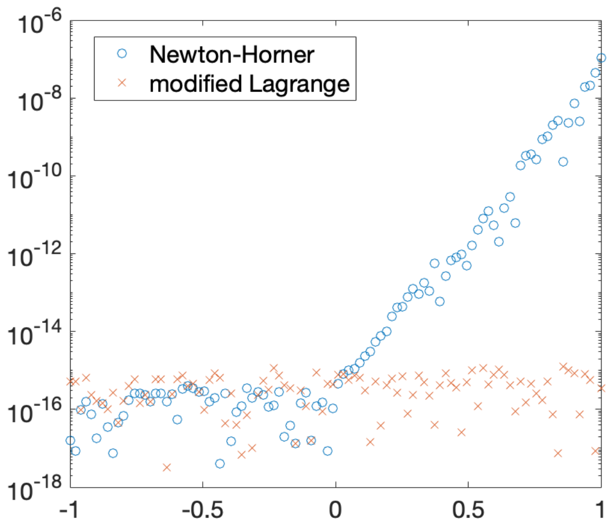
This still costs $O(n)$ operations for each x, and hence is superior to Neville's method when the interpolant needs to be evaluated at a lot of x points.

Stability (optional). Newton's interpolation formula may not always be stable, and sometimes vulnerable to round-off errors. Nick Higham from University of Manchester has discussed the stability of some interpolation formulae including the issue of Newton's, in his paper *The numerical stability of barycentric Lagrange interpolation*. One example given by Higham is as follows; saved in `stability.m`

```
n = 29; x = linspace(0,1,n+1);
x = cos(pi*x); x = flip(x);
y = 1./(1+25*x.^2); h=2*10^3*eps;
xx = linspace(-1+h,1-h,100);
A = divdif(x,y);
yy = newtoninterp(x,diag(A),xx);
yl = lagrange(x,y,xx);

% do again in vpa
digits(50);
X = linspace(vpa(0),vpa(1),n+1);
X = cos(pi*X); X = flip(X);
Y = 1./(1+25*X.^2);
H = 2*vpa(10)^3*vpa(eps);
XX = linspace(-vpa(1)+H,vpa(1)-H,100);
YY = lagrange(X,Y,XX);

% compare
semilogy(xx,abs(yy-YY)./abs(YY),'o')
hold on
semilogy(xx,abs(yl-YY)./abs(YY),'x')
```


Newton-Horner
modified Lagrange