A TOOLBOX AND SOME EXAMPLES

- First of all, we need a simple MATLAB function that generates the zeros of $T_n(x)$ in $[-1,+1]$:

```
function z=ChebyZeros(n)

z=-cos((2*(1:n)'-1)/(2*n)*pi);
```

- The function generates a column vector z that contains the n zeros of T_n in $[-1,+1]$, sorted in ascending order. Here is the output for $n=5$:

```
>> ChebyZeros(5)
ans =
   -0.9511
   -0.5878
   -0.0000
    0.5878
    0.9511
```

- The zeros in *[-1,+1]* are however not enough; in general, we need also the nodes remapped into a general interval of the real line:

```
function [z,x]=ChebyNodes(a,b,n)

z=ChebyZeros(n);
x=(z+1)*((b-a)/2)+a;
```

- The numbers *a* and *b* correspond to the extremes of *[a,b]*, and *n* is the needed number of nodes.
- The function generates two vectors: *z* contains the *n* zeros of T_n in *[-1,+1]*, while *x* contains the corresponding *n* zeros of T_n in *[a,b]*.

```
>> [z,x]=ChebyNodes(1,3,5)
z =
   -0.9511
   -0.5878
   -0.0000
    0.5878
    0.9511
x =
    1.0489
    1.4122
    2.0000
    2.5878
    2.9511
```

- The next step is a function that evaluates $T_n(x)$ at a given point in $[-1,+1]$.
- More generally, we need a function able to evaluate $T_j(x)$ for $j=0,1,\ldots,d$ at given points in $[-1,+1]$.
- Moreover, the function should also be able to evaluate the k-fold tensor product of $\{T_j\}$ at given points in $[-1,+1]^k$.
- The inputs are:
  - the $m \times k$ matrix x, containing the m points in $[-1,+1]^k$ where to evaluate the the k-fold tensor product of T_j (each row of x contains the coordinates of a single point in $[-1,+1]^k$);
  - the integer d, which corresponds to the maximum degree to be evaluated.

```
function T=Cheby(x,d)

[m,k]=size(x);
d=round(d);
x=acos(x);
D=0:d;
D=D(ones(m,1),:);
if k==1
    T=cos(D.*x(:,ones(1,d+1)));
else
    C=zeros(m,d+1,k);
    for j=1:k
        C(:,:,j)=cos(D.*x(:,j*ones(1,d+1)));
    end
    T=C(:,:,k);
    q=(1:d+1)’;
    for i=k-1:-1:1
        z=repmat(T,1,d+1);
        q1=q(:,ones(1,size(T,2)))’;
        T=C(:,q1(:),i).*z;
    end
end
```

In the univariate case, i.e. when $k = 1$, the function generates the following matrix:

$$T = \begin{bmatrix} T_0(x_1) & T_1(x_1) & \cdots & T_d(x_1) \\ T_0(x_2) & T_1(x_2) & \cdots & T_d(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ T_0(x_m) & T_1(x_m) & \cdots & T_d(x_m) \end{bmatrix}$$

Note that, if $c \in R^{d+1}$ is a column vector of coefficients, then:

$$T \bullet c = \begin{bmatrix} \sum_{j=0}^{d} c_j T_j(x_1) \\ \sum_{j=0}^{d} c_j T_j(x_2) \\ \vdots \\ \sum_{j=0}^{d} c_j T_j(x_m) \end{bmatrix} = \begin{bmatrix} h_d(x_1) \\ h_d(x_2) \\ \vdots \\ h_d(x_m) \end{bmatrix}$$

In the bivariate case, i.e. when $k = 2$, the program constructs two intermediate matrices, one for each column of x:

$$T_1 = \begin{bmatrix} T_0(x_{11}) & \cdots & T_d(x_{11}) \\ \vdots & \ddots & \vdots \\ T_0(x_{m1}) & \cdots & T_d(x_{m1}) \end{bmatrix}, \quad T_2 = \begin{bmatrix} T_0(x_{12}) & \cdots & T_d(x_{12}) \\ \vdots & \ddots & \vdots \\ T_0(x_{m2}) & \cdots & T_d(x_{m2}) \end{bmatrix}$$

Then, the algorithm computes the Kronecker tensor product between each row of T_1 and the corresponding row of T_2; hence, the final output becomes:

$$T = \begin{bmatrix} T_0(x_{11})T_0(x_{12}) & T_0(x_{11})T_1(x_{12}) & \cdots & T_d(x_{11})T_d(x_{12}) \\ T_0(x_{21})T_0(x_{22}) & T_0(x_{21})T_1(x_{22}) & \cdots & T_d(x_{21})T_d(x_{22}) \\ \vdots & \vdots & \ddots & \vdots \\ T_0(x_{m1})T_0(x_{m2}) & T_0(x_{m1})T_1(x_{m2}) & \cdots & T_d(x_{m1})T_d(x_{m2}) \end{bmatrix}$$

Note that, if $c \in R^{(d+1)^2}$ is a column vector of coefficients, then:

$$T \bullet c = \begin{bmatrix} \sum_{i=0}^{d} \sum_{j=0}^{d} c_{ij} T_i(x_{11}) T_j(x_{12}) \\ \sum_{i=0}^{d} \sum_{j=0}^{d} c_{ij} T_i(x_{21}) T_j(x_{22}) \\ \vdots \\ \sum_{i=0}^{d} \sum_{j=0}^{d} c_{ij} T_i(x_{m1}) T_j(x_{m2}) \end{bmatrix} = \begin{bmatrix} h_d(x_{1,\cdot}) \\ h_d(x_{2,\cdot}) \\ \vdots \\ h_d(x_{m,\cdot}) \end{bmatrix}$$

In general, the output matrix T will be a $m \times (d+1)^k$ matrix such that:

$$T \bullet c = \begin{bmatrix} \sum_{i=0}^{d} \sum_{j=0}^{d} \dots \sum_{z=0}^{d} c_{ij\dots z} T_i(x_{11}) T_j(x_{12}) \dots T_j(x_{1k}) \\ \sum_{i=0}^{d} \sum_{j=0}^{d} \dots \sum_{z=0}^{d} c_{ij\dots z} T_i(x_{21}) T_j(x_{22}) \dots T_j(x_{2k}) \\ \vdots \\ \sum_{i=0}^{d} \sum_{j=0}^{d} \dots \sum_{z=0}^{d} c_{ij\dots z} T_i(x_{m1}) T_j(x_{m2}) \dots T_j(x_{mk}) \end{bmatrix}$$

for a suitable $c \in (d+1)^k$ column vector of coefficients.

The output for the $m = 5$, $d = 5$ case is following:

```
>> x=chebyzeros(5); cheby(x,5)
ans =
    1.0000   -0.9511    0.8090   -0.5878    0.3090    0.0000
    1.0000   -0.5878   -0.3090    0.9511   -0.8090   -0.0000
    1.0000   -0.0000   -1.0000    0.0000    1.0000   -0.0000
    1.0000    0.5878   -0.3090   -0.9511   -0.8090    0.0000
    1.0000    0.9511    0.8090    0.5878    0.3090   -0.0000
```

- We need now a function that computes the value of:

$$h_d(x,\mathbf{c}) = \sum_{j=0}^{d} c_j T_j(x)$$

at a given set of points in $[-1,+1]^k$ and for given vector of coefficients c.

- This task is accomplished by the following function *ChebyPol*:

```
function p=ChebyPol(x,c)

d=round(size(c,1)^(1/size(x,2))-1);
p=real(cheby(x,d)*c);
```

- We have now all the necessary tools to apply our projections methods to a simple univariate example.
- We will now use orthogonal collocation, the Galerkin, and Least Squares methods, to approximate the function:

$$f(x) = \cos(x) + \sin(x)$$

over the interval [0,10].

- Approximating a known function enables us to observe the size of the true approximation error.
- Furthermore, approximating a known function can be interpreted as solving the following functional equation:

$$g[f(x)] \equiv f(x) - \cos(x) - \sin(x) = 0$$

- The function to be approximated, *f*, is implemented in the following MATLAB function:

```
function f=TestFun(x)

f=cos(x)+sin(x);
```

- This function will be approximated by:

$$h_d(x,\mathbf{c}) = \sum_{j=0}^{d} c_j T_j(x)$$

- The orthogonal collocation method solves for ***c*** the following system of $d+1$ nonlinear equations:

$$\cos(x_k) + \sin(x_k) - h_d(x_k,\mathbf{c}) = 0, \quad k = 1,2,\ldots,d+1$$

where the x_k are the zeros of T_{d+1} in $[0,10]$.

- Hence, given a $(d+1)\times 1$ vector x, the $(d+1)\times(d+1)$ corresponding matrix T, and a $(d+1)\times 1$ vector of initial coefficients c, we have to solve the following “residual function:”

```
function res=ResFunCol(c,x,T)

f=testfun(x);
h=T*c;
res=f-h;
```

- The procedure to approximate $f(x)$ over $[0,10]$ with an algebraic polynomial in the T_j of degree 5 is implemented in the following script:

```
a=0;
b=10;
d=5;
opt=[sqrt(eps) 1e-7 500 2];
tic
cf0=[1;0.2];
for n=2:d+1
    [z,x]=chebynodes(a,b,n);
    T=cheby(z,n-1);
    cfc=newtonsolve(’ResFunCol’,cf0,opt,x,T);
    cf0=NewGuess(cfc,1);
end
toc
```

- Using a **continuation** approach, the code starts by solving the simplest problem, i.e. a linear approximation, and then uses the result as the initial guess for the quadratic approximation problem.

- The solution to the quadratic problem is then used as the initial guess for the cubic one, and so on.

- Of course, the coefficient vector for a d-degree approximation, c_d, has only $d+1$ elements, while the initial guess for the $(d+1)$-degree approximation, c_{d+1}, needs $d+2$ elements: the missing element is simply assumed to be equal to 1/10 of the last coefficient in c_d:

$$\mathbf{c}_{d+1} = [c_0, c_1, \ldots, c_d, c_d/10]$$

- In the **bivariate** case, things are slightly more complicated. Compare $h_1(x,\boldsymbol{c})$ and $h_2(x,\boldsymbol{c})$:

$$\begin{aligned} h_1(x,\mathbf{c}) &= c_{00} + c_{01}T_1(x_2) + c_{10}T_1(x_1) + c_{11}T_1(x_1)T_1(x_2) \\ h_2(x,\mathbf{c}) &= c_{00} + c_{01}T_1(x_2) + c_{02}T_2(x_2) + \\ &\quad c_{10}T_1(x_1) + c_{11}T_1(x_1)T_1(x_2) + c_{12}T_1(x_1)T_2(x_2) + \\ &\quad c_{20}T_2(x_1) + c_{21}T_2(x_1)T_1(x_2) + c_{22}T_2(x_1)T_2(x_2) \end{aligned}$$

- If $\hat{\boldsymbol{c}}_1$ is the coefficient vector for the bivariate linear approximation, then the intial guess for the successive quadratic approximation will be:

$$\mathbf{c}_2 = [c_{00}, c_{01}, 0, c_{10}, c_{11}, 0, 0, 0, 0]$$

- Evidently, adding zeros at the end of the vector is not enough.

- In general, the scheme can be really complicated. However, the function *NewGuess* takes care of upgrading the initial guess at each iteration:

```
function cf0=NewGuess(cf,k)

[n,m]=size(cf);
k=round(k);
if k==1
 cf0=[cf;cf(end,:)/10];
else
 z=round(n^(1/k));
 q=round(z^(k-1));
 cf1=NewGuess(cf(1:q,:),k-1);
 for j=2:z
 cf1=[cf1;NewGuess(cf((1+(j-1)*q:j*q),:),k-1)];
 end
 cf0=[cf1;zeros(round((z+1)^(k-1)),m)];
end
```

- The Galerkin method can now be easily implemented.
- We just need to rewrite the “residual function:”

```
function res=ResFunGal(cf,x,T)

res=T’*ResFunCol(cf,x,T);
```

- Furthermore, we also need to modify the main script:

```
m=30;
tic
cf0=[1;0.2];
[z,x]=ChebyNodes(a,b,m);
for n=2:d+1
    T=Cheby(z,n-1);
    cfg=NewtonSolve(’ResFunGal’,cf0,opt,x,T);
    cf0=NewGuess(cfg,1);
end
toc
```

- Implementing the Least Squares method is even simpler:

```
tic
cf0=[1;0.2];
[z,x]=ChebyNodes(a,b,m);
for n=2:d+1
    T=Cheby(z,n-1);
    cfq=GaussNewtonMin('ResFunCol',cf0,[],x,T);
    cf0=NewGuess(cfq,1);
end
toc
```

- The following script computes the mean, median, and max of the absolute approximation error, and its standard deviation, over 1000 equally spaced points in *[0,10]*, and plots the results:

```
[z,x]=UniformNodes(a,b,1000);
f=TestFun(x);
hc=ChebyPol(z,cfc);
hg=Chebypol(z,cfg);
hq=ChebyPol(z,cfq);
er=[f-hc,f-hg,f-hq];
disp([’Avg. Abs. Er.: ’ num2str(mean(abs(er)))])
disp([’Avg. Med. Er.: ’ num2str(median(abs(er)))])
disp([’Std. Er.:      ’ num2str(std(er))])
disp([’Max. Abs. Er.: ’ num2str(max(abs(er)))])
subplot(2,1,1), plot(x,[f,hc,hg,hq])
legend(’True’,’Col’,’Gal’,’Lsq’,0), ylabel(’f(x)’)
subplot(2,1,2), plot(x,er)
legend(’Col’,’Gal’,’Lsq’,0), ylabel(’App. error’)
```

d=5

```
>> chebytest
Elapsed time is 0.000000 seconds.
Elapsed time is 0.015000 seconds.
Elapsed time is 0.016000 seconds.
Avg. Abs. Er.: 0.137      0.14044     0.14044
Avg. Med. Er.: 0.090708  0.13388     0.13388
Std. Er.:      0.17979   0.16376     0.16376
Max. Abs. Er.: 0.37866   0.28436     0.28436
```

d=15

```
>> chebytest
Elapsed time is 0.015000 seconds.
Elapsed time is 0.031000 seconds.
Elapsed time is 0.063000 seconds.
Avg. Abs. Er.: 6.6927e-008 6.8165e-008 6.8165e-008
Avg. Med. Er.: 6.2198e-008 7.2494e-008 7.2494e-008
Std. Er.:      7.8082e-008 7.6638e-008 7.6638e-008
Max. Abs. Er.: 1.5554e-007  1.296e-007  1.296e-007
```

d=20

```
>> chebytest
Elapsed time is 0.016000 seconds.
Elapsed time is 0.031000 seconds.
Elapsed time is 0.078000 seconds.
Avg. Abs. Er.: 5.7022e-010 6.2445e-010 2.6697e-008
Avg. Med. Er.: 3.719e-010   5.81e-010 2.6215e-008
Std. Er.:      7.6503e-010 7.4391e-010 3.0648e-008
Max. Abs. Er.: 2.1792e-009 1.5478e-009 5.9076e-008
```

- The previous example can be easily extended to the multivariate case.
- We will now use orthogonal collocation, the Galerkin, and Least Squares methods, to approximate the function:

$$f(x) = \cos(x_1) + \sin(x_2)$$

over the rectangle $[0,10]^2$.

- The function to be approximated is implemented as:

```
function f=TestFun2(x)

f=cos(x(:,1))+sin(x(:,2));
```

- This function will be approximated by:

$$h_d(x,\mathbf{c}) = \sum_{i=0}^{d}\sum_{j=0}^{d} c_{ij} T_i(x_1) T_j(x_2)$$

- The orthogonal collocation method solves for ***c*** the following system of $(d+1)^2$ nonlinear equations:

$$\cos(x_{1k}) + \sin(x_{2k}) - h_d(x_k,\mathbf{c}) = 0, \quad k = 1,2,\ldots,(d+1)^2$$

- In the univariate case, the collocation nodes x_k were the zeros of T_{d+1} in $[0,10]$.
- Now, we have to optimally select $(d+1)^2$ points in the rectangle $[0,10]^2$, in order to minimize the approximation error.

- We start by obtaining the $d+1$ zeros of T_{d+1} over the approximation intervals for x_1 and x_2; being both intervals equal, we just need $\{x_j\}$, i.e. the zeros of T_{d+1} in $[0,10]$.

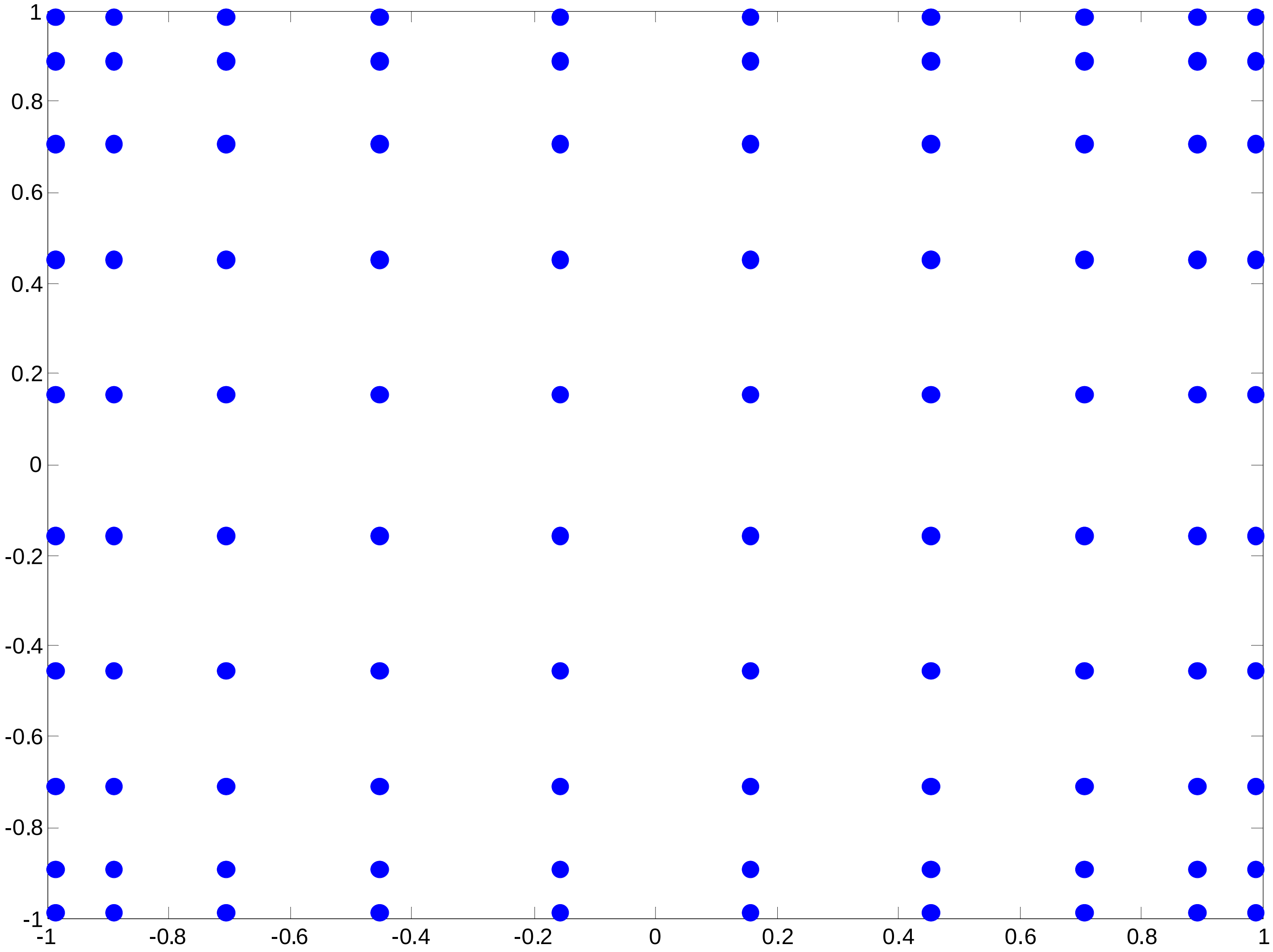
- Then, we build the set of all possible combinations of the elements of $\{x_j\}$, i.e. $\{x_i, x_j\}$, where $i=1,2,\ldots, d+1$ and $j=1,2,\ldots,d+1$. In other words, we build the set $x=\{x_j\} \otimes \{x_j\}$:

$$x = \begin{bmatrix} \hat{x}_1 & \hat{x}_1 \\ \hat{x}_1 & \hat{x}_2 \\ \vdots & \vdots \\ \hat{x}_1 & \hat{x}_{d+1} \\ \hat{x}_2 & \hat{x}_1 \\ \hat{x}_2 & \hat{x}_2 \\ \vdots & \vdots \\ \hat{x}_{d+1} & \hat{x}_{d+1} \end{bmatrix}$$

- Given a vector *v* whose columns contain the zeros of T_{d+1} polynomial over the relevant intervals (in our case, the columns of *v* are both equal to $\{x_j\}$, the zeros of T_{d+1} over *[0,10]*), the set of multivariate collocation nodes can be generated by the following function *Stack*:

```
function s=stack(v)

[n,k]=size(v);
s=v(:,1);
for j=1:k-1
    m=size(s,1);
    q=repmat(1:m,n,1);
    s=[s(q(:),1:j),repmat(v(:,j+1),m,1)];
end
```


1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1

- Hence, given a $(d+1)^2\times 2$ vector x, the $(d+1)^2\times(d+1)^2$ corresponding matrix T, and a $(d+1)^2\times 1$ vector of initial coefficients ***c***, we have to solve the “residual function:”

```
function res=ResFunCol2(cf,x,T)

f=testfun2(x);
h=T*cf;
res=f-h;
```

- The “residual function” for the Galerkin method becomes:

```
function res=ResFunGal2(cf,x,T)

res=T’*ResFunCol2(cf,x,T);
```

Bivariate Orthogonal Collocation

```
a=0;
b=10;
d=5;
opt=[sqrt(eps) 1e-7 500 2];
tic
cf0=[0.6;-0.15;-0.5;0];
for n=2:d+1
    [z,x]=ChebyNodes(a,b,n);
    xv=Stack([x,x]);
    zv=Stack([z,z]);
    T=Cheby(zv,n-1);
    cfc=NewtonSolve(’ResFunCol2’,cf0,opt,xv,T);
    cf0=NewGuess(cfc,2);
end
toc
```

Bivariate Galerkin method

```
m=20;
tic
cf0=[0.6;-0.15;-0.5;0];
[z,x]=ChebyNodes(a,b,m);
for n=2:d+1
    xv=Stack([x,x]);
    zv=Stack([z,z]);
    T=Cheby(zv,n-1);
    cfg=NewtonSolve(’ResFunGal2’,cf0,opt,xv,T);
    cf0=NewGuess(cfg,2);
end
toc
```

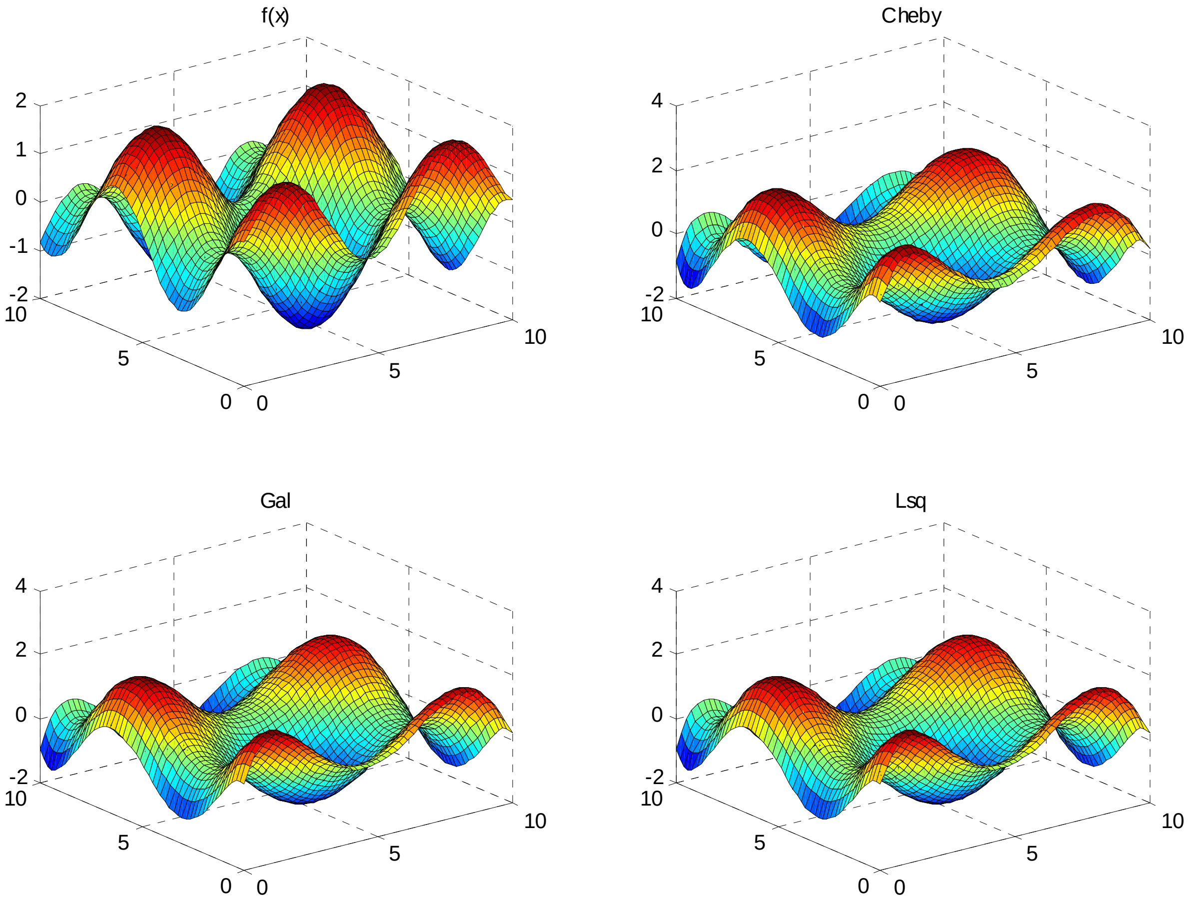
Bivariate Least Squares method

```
tic
cf0=[0.6;-0.15;-0.5;0];
[z,x]=ChebyNodes(a,b,m);
for n=2:d+1
    xv=Stack([x,x]);
    zv=Stack([z,z]);
    T=Cheby(zv,n-1);
    cfq=GaussNewtonMin(’ResFunCol2’,cf0,[],xv,T);
    cf0=NewGuess(cfq,2);
end
toc
```

```
p=50;
[z,x]=UniformNodes(a,b,p);
xv=Stack([x,x]);
zv=Stack([z,z]);
f=TestFun2(xv);
hc=ChebyPol(zv,cfc);
hg=ChebyPol(zv,cfg);
hq=ChebyPol(zv,cfq);
er=[f-hc,f-hg,f-hq];
disp([’Avg. Abs. Er.: ’ num2str(mean(abs(er)))])
disp([’Avg. Med. Er.: ’ num2str(median(abs(er)))])
disp([’Std. Er.:      ’ num2str(std(er))])
disp([’Max. Abs. Er.: ’ num2str(max(abs(er)))])
f=reshape(f,p,p)’;
hc=reshape(hc,p,p)’;
hg=reshape(hg,p,p)’;
hq=reshape(hq,p,p)’;
erc=reshape(er(:,1),p,p)’;
erg=reshape(er(:,2),p,p)’;
erq=reshape(er(:,3),p,p)’;
subplot(2,2,1), surf(x,x,f), title(’f(x)’)
subplot(2,2,2), surf(x,x,hc), title(’Cheby’)
subplot(2,2,3), surf(x,x,hg), title(’Gal’)
subplot(2,2,4), surf(x,x,hq), title(’Lsq’)
pause
subplot(2,2,1), surf(x,x,f), title(’f(x)’)
subplot(2,2,2), surf(x,x,erc), title(’Cheby’)
subplot(2,2,3), surf(x,x,erg), title(’Gal’)
subplot(2,2,4), surf(x,x,erq), title(’Lsq’)
```

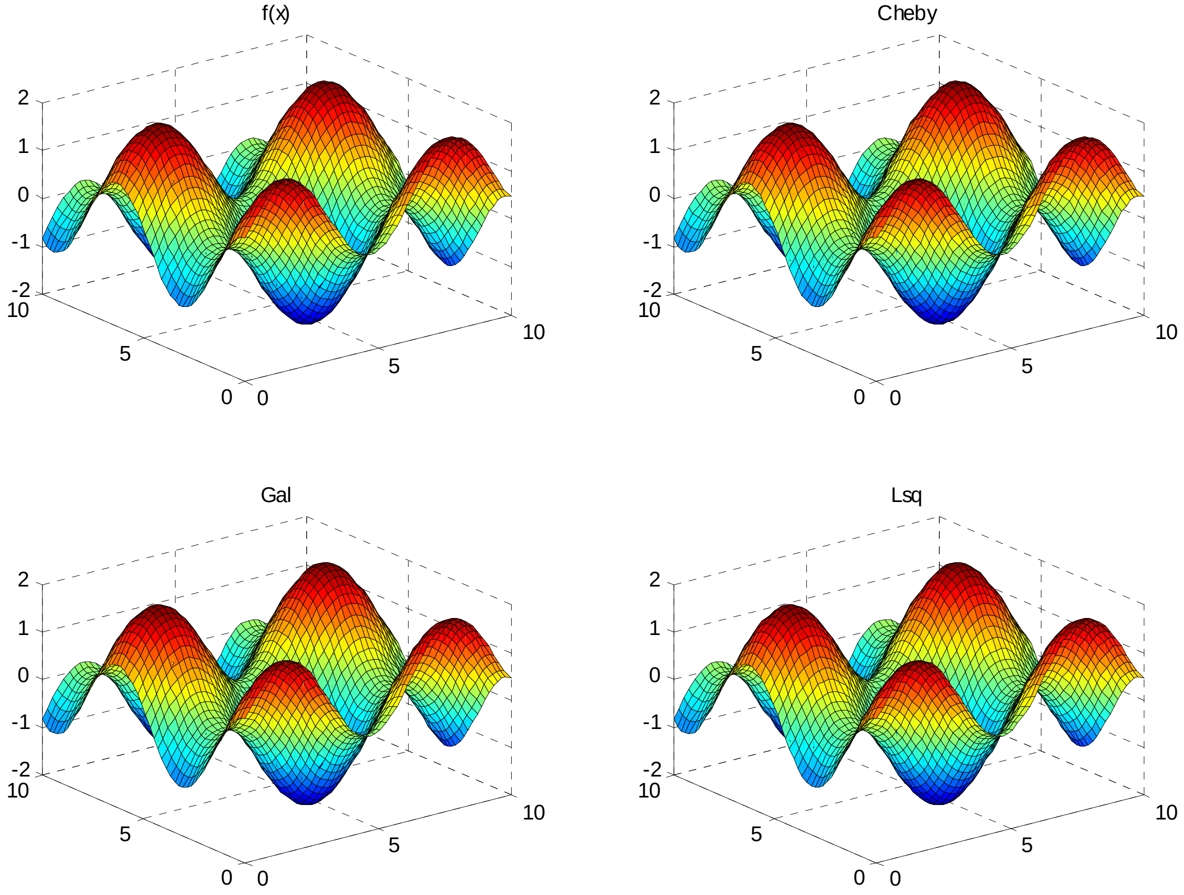
d=5

```
>> chebytest2
Elapsed time is 0.031000 seconds.
Elapsed time is 0.031000 seconds.
Elapsed time is 0.094000 seconds.
Avg. Abs. Er.: 0.19093     0.18143     0.18143
Avg. Med. Er.: 0.18033     0.17011     0.17011
Std. Er.:      0.22723      0.2123      0.2123
Max. Abs. Er.: 0.56687     0.45321     0.45321
```

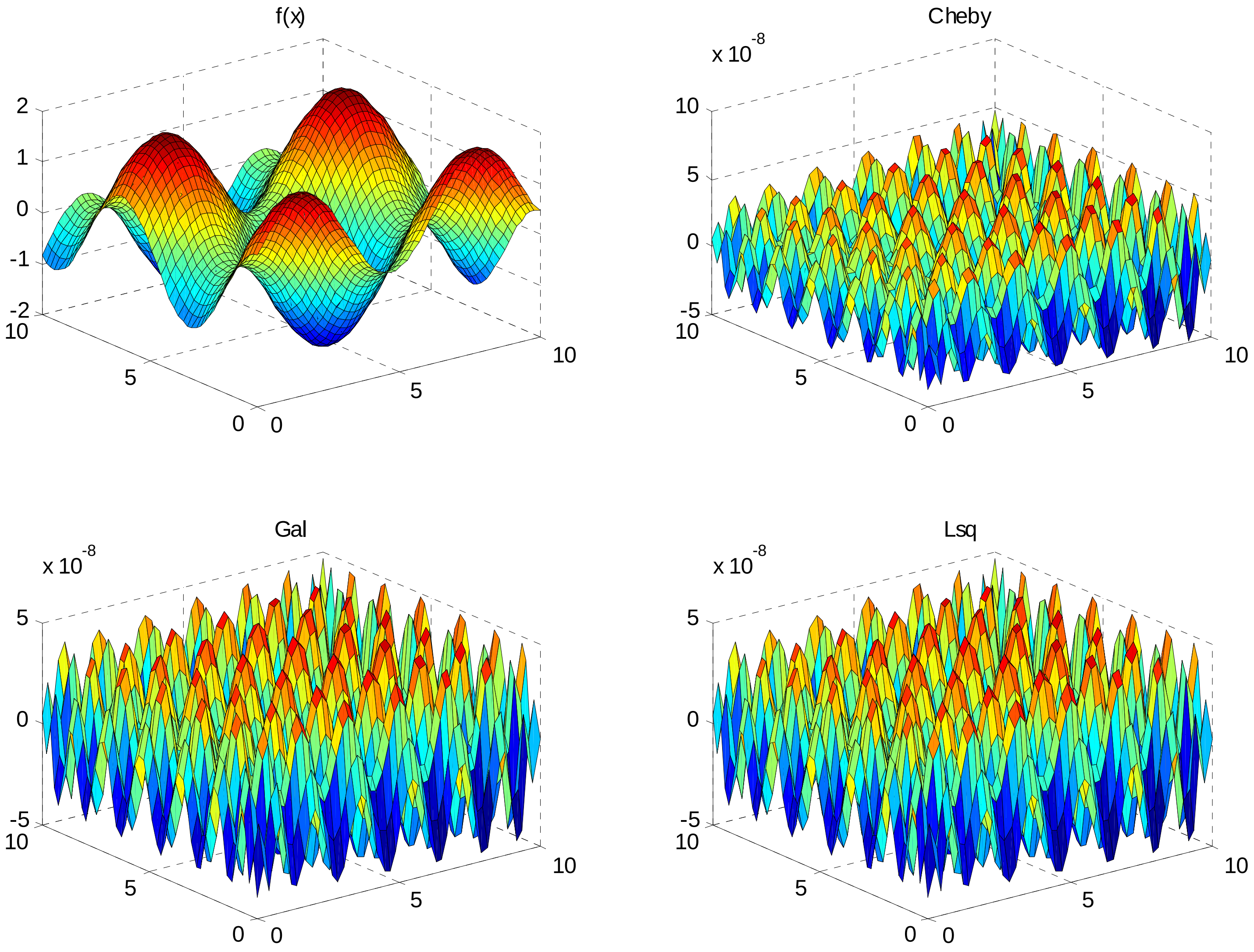


d=20

```
>> chebytest2
Elapsed time is 2.172000 seconds.
Elapsed time is 6.266000 seconds.
Elapsed time is 11.984000 seconds.
Avg. Abs. Er.: 1.8245e-008 1.7973e-008 1.7973e-008
Avg. Med. Er.: 1.6208e-008 1.6203e-008 1.6203e-008
Std. Er.:      2.2379e-008 2.2126e-008 2.2126e-008
Max. Abs. Er.: 5.2241e-008 4.8226e-008 4.8226e-008
```



d=20


f(x)
2
1
0
-1
-2
10
5
0
0
5
10
Cheby
x 10^-8
10
5
0
-5
10
5
0
0
5
10
Gal
x 10^-8
5
0
-5
10
5
0
0
5
10
Lsq
x 10^-8
5
0
-5
10
5
0
0
5
10