DOPbox
Discrete Orthogonal Polynomial Toolbox
Getting Started

Matthew Harker and Paul O'Leary
Institute for Automation
University of Leoben
A-8700 Leoben,Austria
URL: automation.unileoben.ac.at

Original: January 9, 2013
© 2013

Last Modified: June 3, 2013

Contents

1 Toolbox description

This package contains a set of m-files which implement a **D**iscrete **O**rthogonal **P**olynomial toolbox called **DOPbox**. There are many applications of discrete opthogonal polynomials, e.g., in the solution of ordinary differential equations (ODEs), partial differential equations (PDEs), boundary value problems (BVPs), initial value problems (IVPs), inverse-BVPs etc.

This documentation describes the functions briefly and gives examples as to how they are used. The theory behind this toolbox can be found in the following publications: The paper [1]

```
@inproceedings{
oleary2008b,
Author = {O'Leary, P. and Harker, M.},
Title = {An Algebraic Framework for Discrete Basis Functions in Computer Vision},
BookTitle = {2008 $6^{\textrm{th}}$ ICVGIP},
Address= {Bhubaneswar, India},
Publisher = {IEEE},
Pages = {150-157},
Year = {2008} }
```

provides details of the synthesis algorithm and the application of the Gram polynomials to image processing. The paper [2]

```
@inproceedings{oleary2010C,
Author = {O'Leary, P. and Harker, M.},
Title = {Discrete Polynomial Moments and Savitzky-Golay Smoothing},
BookTitle = {Waset Special Journal},
Volume = {72},
DOI = {},
Pages = {439--443},
Year = {2010}}
```

provides information on when the Gram-Schmidt orthogonalization fails and also the implementation of local polynomial approximations. The application of `DOPbox` to BVPs and IVBPs is documented in [3],

```
@article{Oleary2012,
author    = {Paul O'Leary and
Matthew Harker},
title     = {A Framework for the Evaluation of Inclinometer Data in the
Measurement of Structures},
journal   = {IEEE T. Instrumentation and Measurement},
volume    = {61},
number    = {5},
year      = {2012},
pages     = {1237-1251},
ee        = {http://dx.doi.org/10.1109/TIM.2011.2180969}}
```

2 Toolbox Functions

All the m-files contain headers with help for the specific function, see for example:

```
help dopNodes
```

```
Purpose : This function generates a set of n nodes of the requested type.
In general such nodes are then used to synthesize basis functions.

Use (syntax): nodes = <strong>dopNodes</strong>( n, type )

Input Parameters :
      n: the number of nodes
      type: this paramater determines which set of nodes are generated.
      The type should be one of the following:

      'Gram': generate the nodes required for the gram polynomials.
      'GramEnds': a set of equally spaced nodes in the interval [-1,1].
      'Cheby': the Chebyshev nodes.
      'ChebyEnds': the Chebyshev nodes in the full interval [-1,1].

Return Parameters :
      nodes: an n column vector containing the values of the nodes.

Description and algorithms:

References :

Author :  Matthew Harker and Paul O'Leary
Date :    17. January 2013
Version : 1.0

(c) 2013 Matthew Harker and Paul O'Leary,
Chair of Automation, University of Leoben, Leoben, Austria
email: office@harkeroleary.org,
url: www.harkeroleary.org

History:
  Date:           Comment:
```

Clean up prior to running the code

```
clear all;
close all;
```

2.1 Generate a set of nodes – **dopNodes.m**

this function generates a set of nodes which can be used to synthesize a set of basis functions, see `help dopNodes` for details of the types of nodes supported.

The function is used in the following manner

```
noNodes = 10;
%
% generate the Gram and Chebyshev nodes
%
gNodes = dopNodes( noNodes, 'Gram' );
```

```
cNodes = dopNodes( noNodes, 'Cheby' );
%
% Generate the modiied nodes
%
geNodes = dopNodes( noNodes, 'GramEnds' );
ceNodes = dopNodes( noNodes, 'ChebyEnds' );
```

It is important to note that the nodes need to be generated exactly at the points where the constraints are to be applied, in particular for boundary values problems. Both the Gram and Chebyshev nodes do not reach to the end of the support; this would mean that constraints could not be placed exactly at the ends of the support. For this reason modified Gram and Chenyshev nodes are also provided, they are modified so that they start at -1 and end at 1.

```
fig1 = figure;
a(1) = subplot(4,1,1);
plot( gNodes, zeros(size(gNodes)), 'ko', 'MarkerFaceColor', 'k');
hold on;
plot([-1,1],[0,0], 'k');
axis([-1,1,-1,1]);
ylabel('Gram ');
%
a(3) = subplot(4,1,2);
plot( cNodes, zeros(size(gNodes)), 'ko', 'MarkerFaceColor', 'k');
hold on;
plot([-1,1],[0,0], 'k');
axis([-1,1,-1,1]);
ylabel('Cheby');
%
a(2) = subplot(4,1,3);
plot( geNodes, zeros(size(gNodes)), 'ko', 'MarkerFaceColor', 'k');
hold on;
plot([-1,1],[0,0], 'k');
axis([-1,1,-1,1]);
ylabel('GramE');
%
a(4) = subplot(4,1,4);
plot( ceNodes, zeros(size(gNodes)), 'ko', 'MarkerFaceColor', 'k');
hold on;
plot([-1,1],[0,0], 'k');
axis([-1,1,-1,1]);
%
xlabel( 'Support' );
ylabel('ChebyE');
%
linkaxes( a, 'x');
```

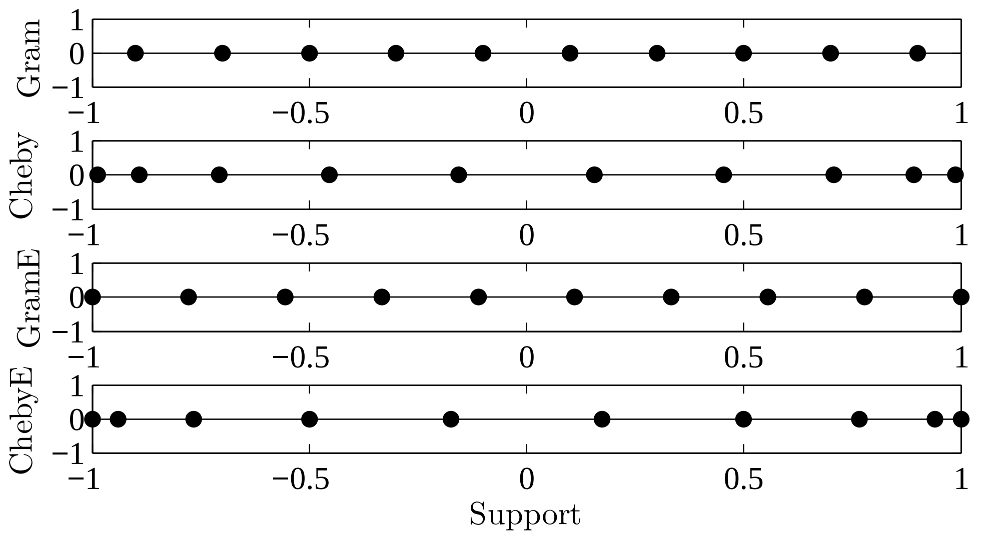


Figure 1: Node placements generated by `dopNodes` with the options for the Gram, Cheby, GramEnds and ChebyEnds.

In the next figure the end of the support is shown.

```
figure(fig1);
axis([-1,-0.65,-1,1]);
```

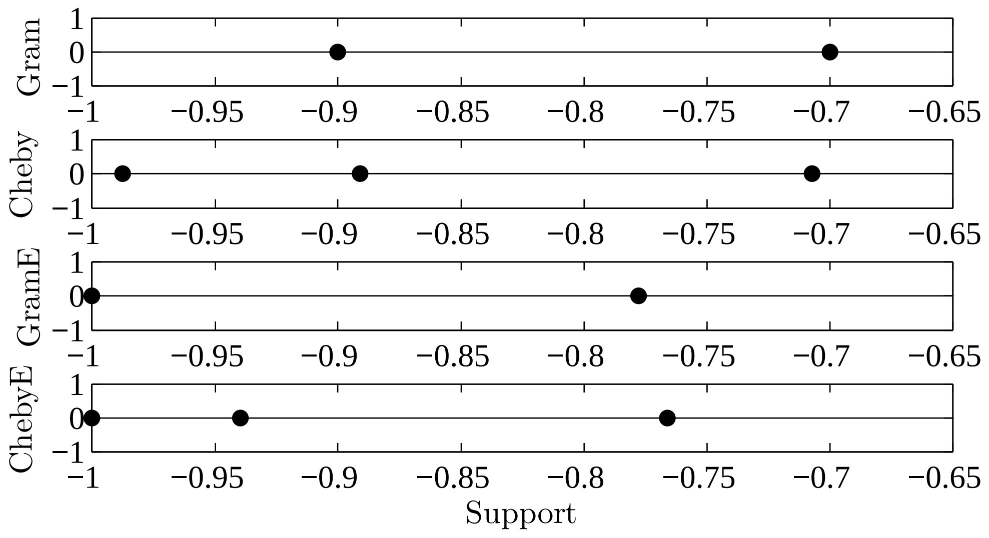


Figure 2: Node placements at the end of the support as generated by `dopNodes` with the options for the Gram, Cheby, GramEnds and ChebyEnds.

2.2 Generate a set of discrete orthogonal polynomials – `dop.m`

The function `dop.m` generates a set of discrete orthogonal polynomials. It has multiple options (see `help dop` for details), the most important features are: the fuinction generates a set of ortho-normal basis functions from a set of nodes. It can also optionally generate the differentials of the basis functions and the coefficients for later interpolation with the basis functions.

The generation of a set of Gram polynomials from the Gram nodes is now demonstrated. A large number of nodes are used to show the nature of the basis functions.

```
noNodes = 100;
noBfs = noNodes;
%
% generate a complete set of basis functions.
%
gNodes = dopNodes( noNodes, 'Gram' );
B = dop( gNodes, noBfs );
%
fig2 = figure;
for k=1:5
   plot( gNodes, B(:,k), 'k' );
   hold on;
end;
axis([-1,1,-0.3, 0.3]);
grid on;
xlabel('Support');
ylabel('Value');
```

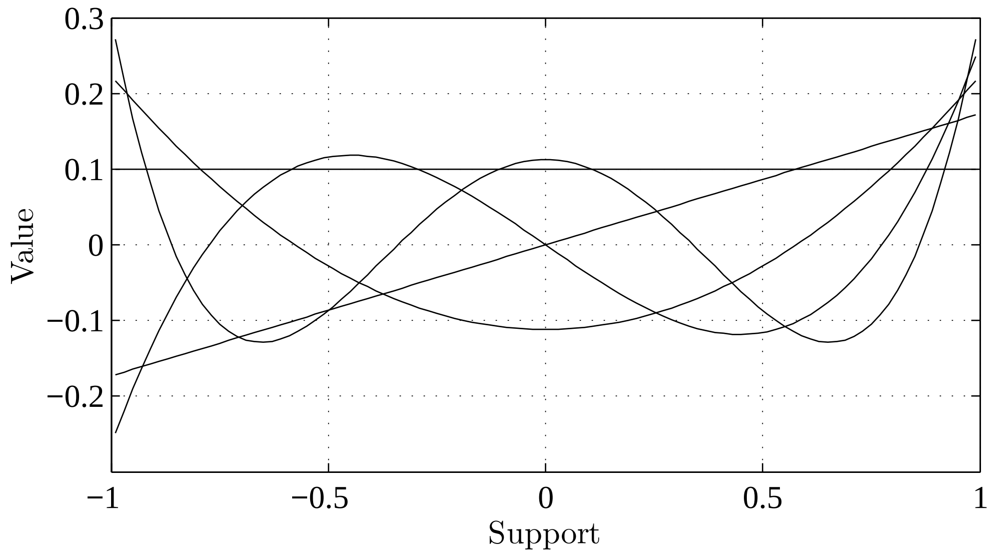


Figure 3: The first 5 Gram basis functions.

2.2.1 Properties of the Basis functions

The basis functions $\boldsymbol{B}$ generated by this function are ortho-normal, i.e. the Gram matrix for the basis functions

$$\boldsymbol{G} \triangleq \boldsymbol{B}^{\mathrm{T}} \boldsymbol{B} = \boldsymbol{I} \tag{1}$$

should be the identity matrix. Consequently, the projection onto the orthogonal complement of $\boldsymbol{G}^{\perp} = \boldsymbol{G} - \boldsymbol{I} = \boldsymbol{0}$ should be exactly a matrix of zeros. However, due to numerical errors this is not the case. $\boldsymbol{G}^{\perp}$ can be inspected to see the quality of the basis functions.

```
G = B' * B;
Gperp = G - eye( noBfs );
%
% Now visualizing the matrix Gperp,
%
xScale = (1:noBfs) - 1;
yScale = xScale;
fig3 = figure;
imagesc( xScale, yScale, Gperp );
colorbar;
xlabel('Degree d');
ylabel('Degree d');
axis image;
```

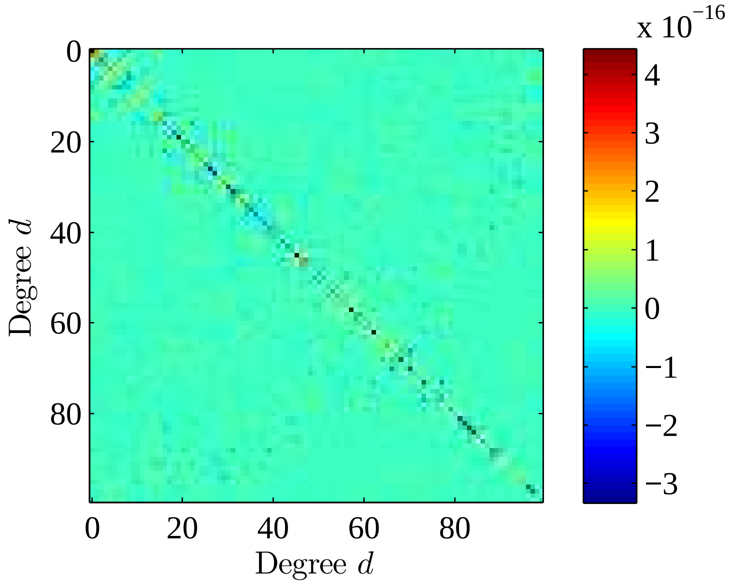


Figure 4: The projection onto the orthogonal complement of the Gram matrix, computed for a set of 100 Gram basis functions. The matrix should be exactly zeror.

Ideally the matrix $\boldsymbol{G}^{\perp}$ should be exactly zero. The Frobenius norm $\|\boldsymbol{G}\|_f$, i.e. the sum of the squares of all entries, can be used as an estimate for the quality of the basis functions.

```
fNormGperp = norm( Gperp, 'fro')
fNormGperp =
   2.9765e-15
```

The number of significant digits can be estimated as $n_s = -\log_{10}(\|\boldsymbol{G}\|_f)$

```
ns = - log10( fNormGperp )
```

```
ns =
   14.5263
```

As can be seen the error is very small. Consequently, the quality of the basis functions is very high.

2.2.2 Generating complex basis functions

Consider the case of wishing to generate a set of basis functions from an arbitrary set of nodes in the complex plane. This, for example, may be desirable when wishing to modell an damped oscillator.

In the example presented here we wish to generate a set of complex basis functions with a predefined envelope. The complex basis functions should be in quadrature to each other, this insures local shift invariance.

```
load arbitraryData;
%
n = length(r);
x = linspace(0,1,n)';
%
figure;
plot( x, r, 'k');
grid on;
xlabel('Support');
ylabel('Envelope');
```

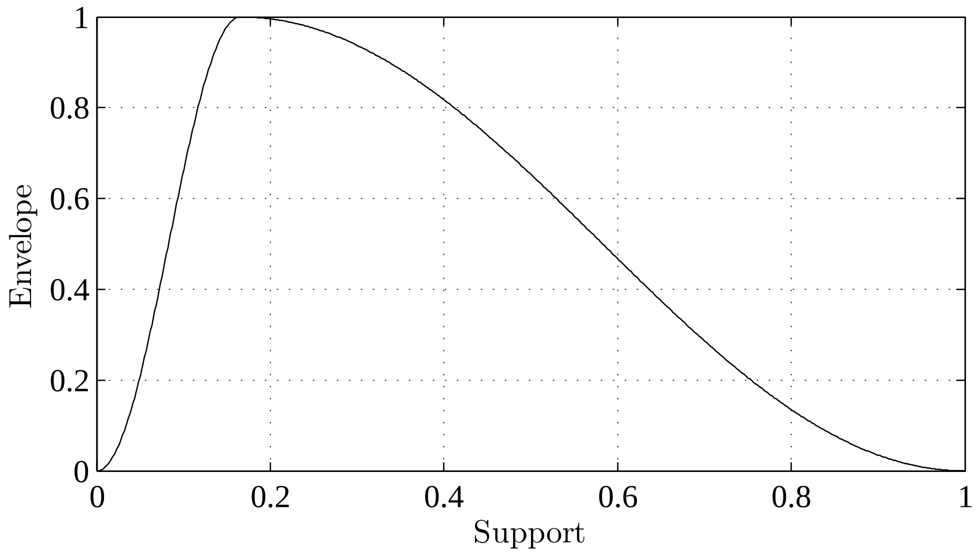


Figure 5: Envelope for the desired basis functions

Now defining the number of cycles required for the basis functions

```
noCycles = 10;
%
% and compute the corresponding angles
%
phi = linspace(0,2*pi*noCycles,n )’;
%
% Compute the rean and imaginary components of the node placements
%
nr = r.*cos( phi );
ni = r.*sin( phi );
n = nr + i * ni;
%
figure;
plot( nr, ni, ’k’);
hold on;
plot( nr, ni, ’k.’);
grid on;
xlabel(’Real’);
ylabel(’Imaginary’);
axis equal;
```

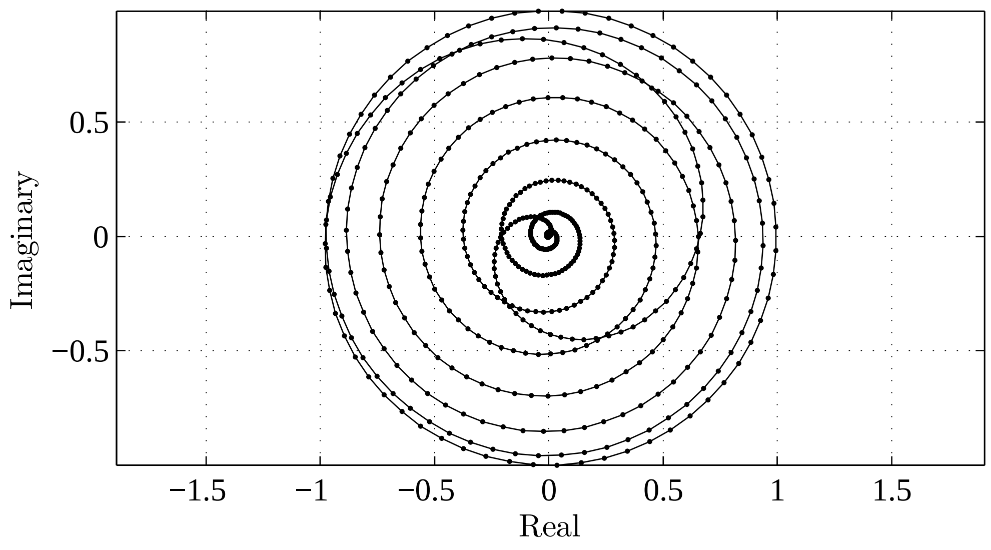
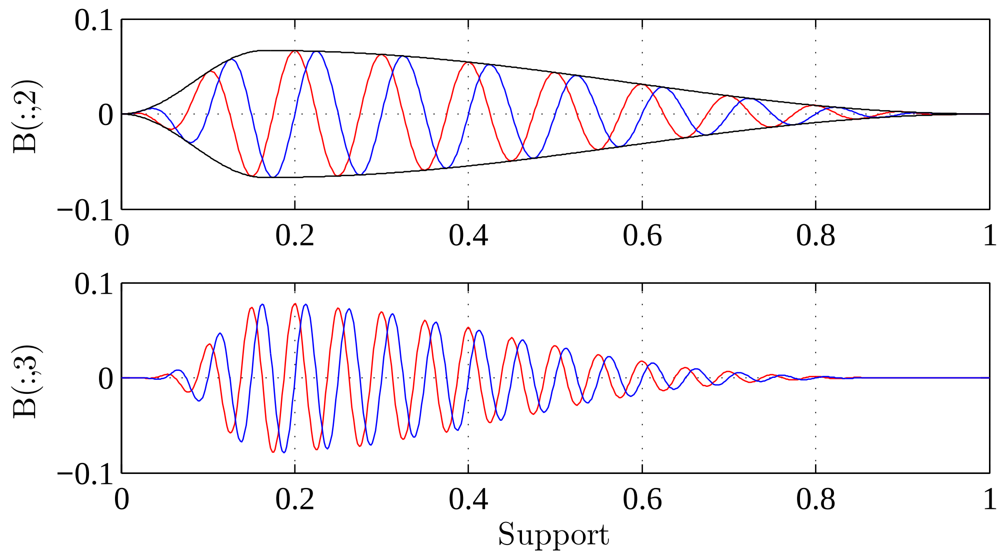


Figure 6: Node placements for the synthesis of the complex basis functions.

Now synthesize the basis functions (only the first 3 are computed for simplicity here.

```
noBfs = 3;
B = dop( n, noBfs );
%
figure;
subplot(2,1,1);
plot( x, real(B(:,2)) , 'r');
hold on;
plot( x, imag(B(:,2)) , 'b');
grid on
ylabel( 'B(:,2)');
plot(x,r/norm(r),'k');
plot(x,-r/norm(r),'k');
%
subplot(2,1,2);
plot( x, real(B(:,3)) , 'r');
hold on;
plot( x, imag(B(:,3)) , 'b');
grid on
ylabel( 'B(:,3)');
%
xlabel('Support');
%\caption{Sets of complex basis functions synthesized using \lstinline{
        dop}}
```



2.2.3 Using **dop.m** to compute a regularizing differential operator.

In this example a polynomial is used, since the function and its derivatives can be computed analytically. This permits an objective comparison of the numerically computed and the analytical derivatives.

```

% Generate a polynomial function for test purposes:
noPts = 100;
x = linspace( -1, 1, noPts )';
p = [5,0,-5,0,-5,0];
dp = polyder( p );
y = polyval( p, x );
dy = polyval( dp, x );
%
% Plot the analytical polynomials and their derivatives together with
        the
% numerically computed values
%
fig4 = figure;
subplot(2,1,1);
plot( x, y, 'k');
ylabel( 'y(x)' );
grid on;
%
subplot(2,1,2);
plot( x, dy, 'k');
ylabel( 'd y(x)/dx' );
grid on;
```

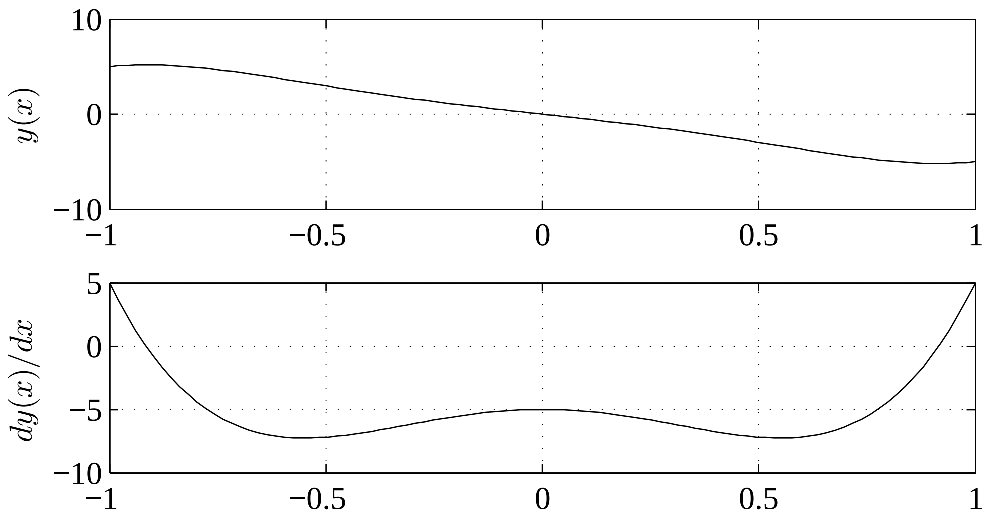


Figure 7: the test function and its analytical derivative

Now investigating the properties of numerical approximations to derivatives: the MATLAB matrix implicitly used to numerically compute derivatives can be extracted as follows:

```
[Dx, Dy] = gradient( eye(noPts));
%
% correct for the step size
%
h = x(2) - x(1);
DyMat = Dy / h ;
%
% and using this matrix to estimate the derivative of y(x).
%
dyMAT = DyMat * y ;
```

The function dop.m can also generate a derivative of the basis functions. Given the basis functions $\boldsymbol{B}$ and their derivatives $\dot{\boldsymbol{B}}$ a differential operator $\boldsymbol{D}$ can be computed:

$$\dot{\boldsymbol{B}} = \boldsymbol{D}\,\boldsymbol{B} \tag{2}$$

consequently

$$\boldsymbol{D}_r \triangleq \boldsymbol{D}\,\boldsymbol{B}\,\boldsymbol{B}^{\mathrm{T}} = \dot{\boldsymbol{B}}\,\boldsymbol{B}^{\mathrm{T}}. \tag{3}$$

This is a regularizing differential operator, it is the differential operator $\boldsymbol{D}$ projected onto the basis functions $\boldsymbol{B}$

```
[B, dB] = dop( x, length(p) );
Dr = dB * B';
%
% and using D to estimate the derivatives
%
```

```
dyDop = Dr * y;
%
fig5 = figure;
plot( x, dy - dyMAT, 'k');
hold on;
plot( x, dy - dyDop, 'r');
grid on;
xlabel('Support');
ylabel('Error in d y(x)');
legend('Matlab Gradient','dop derivative','location','NorthWest');
```

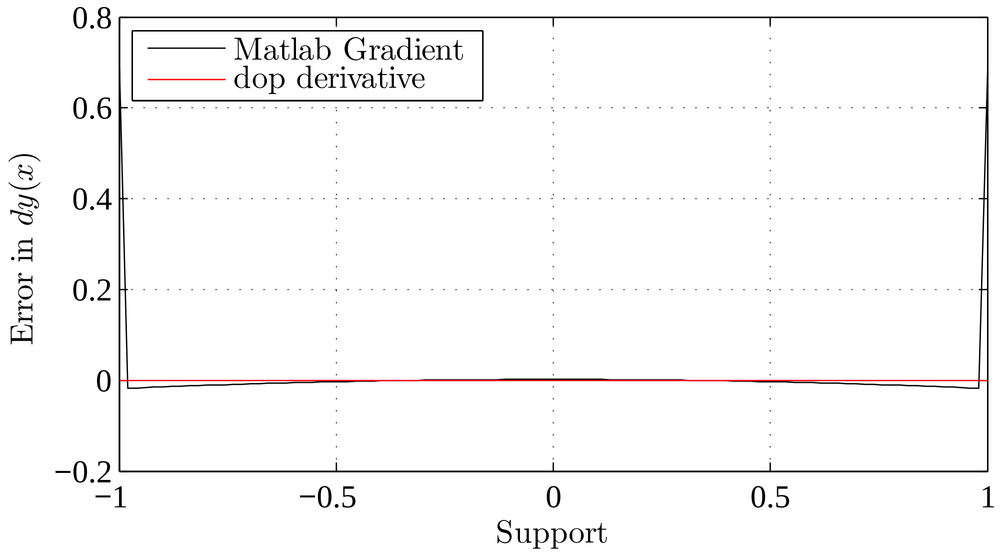


Figure 8: Comparison in the analytical and numericall computed derivatives, using MATLA `gradient` function and the derivative matrix $\boldsymbol{D}$ computed using the `DOPbox`.

Note there are significant error in the numerical compuation of the derivatives using the gradient function, i.e. with a 3 point approximation. The result is grossly incorrect at the ends of the support. This indicates that the classical 3 term differential estimate is not suitable for use with boundary value problems. Zooming in on the plot, shows that there are significant errors over the complete support in the derivatives computed using the 3 term estimate.

In contrast the derivatives computed using the matrix determined from the basis functions and their derivatives shows no significant errors. The matrix $\boldsymbol{D}$ is well suited for the estimation of derivatives.

```
figure(fig5);
axis([-0.95,0.95,-0.02,0.01]);
```

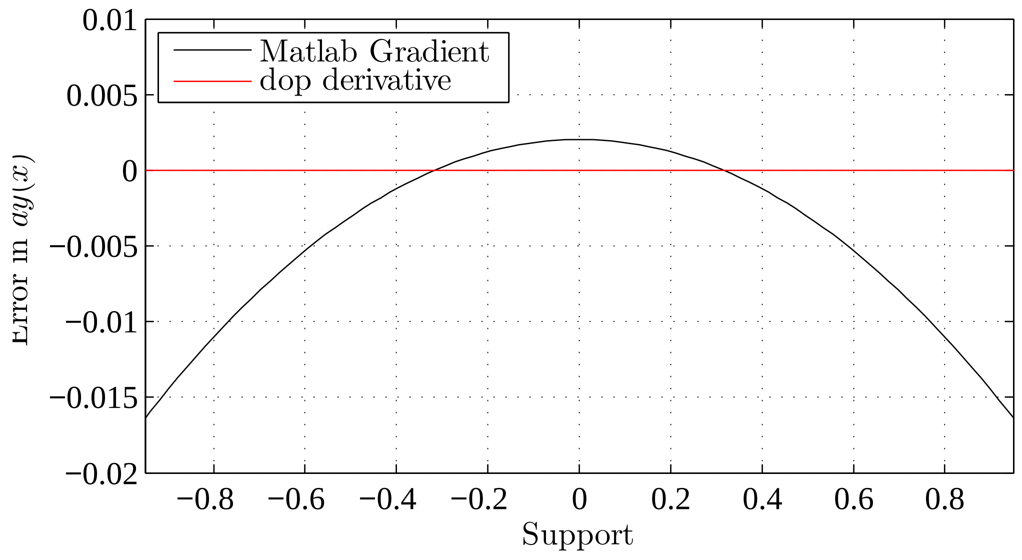


Figure 9: Errors in the estimated derivatives in the centre region of the support.

The strength of the regularizing differentiating matrix is best seen when estimating derivatives in the presence of noise.

```
noiseGain = 0.05;
noise = noiseGain * norm(x) * randn( size(x));
yn = y + noise;
%
dyNMat = DyMat * yn;
dyNDop = Dr * yn;
%
fig5 = figure;
plot( x, dy - dyNMat, 'k');
hold on;
plot( x, dy - dyNDop, 'r');
grid on;
xlabel('Support');
ylabel('Error in d y(x)');
legend('Matlab Gradient','dop derivative','location','NorthWest');
```

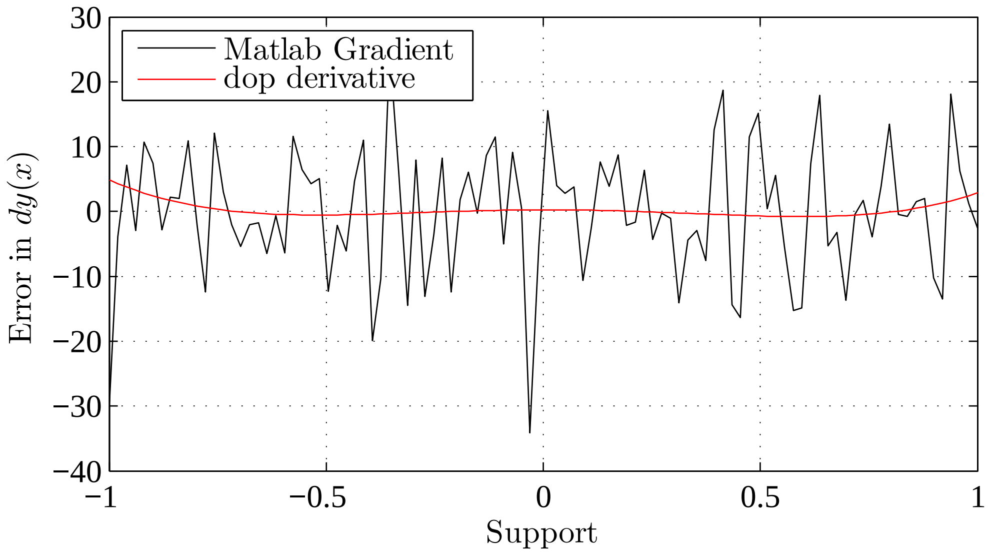


Figure 10: Estimation of derivatives in the presence of noise.

2.3 Generate a matrix $\boldsymbol{D}_L$ to perform local derivative approximation — `dopDiffLocal.m`

This function generates a local differentiating matrix. It supports both sparse and full matrix formats. The full matrix format is the default. Selecting the sparse option will generate a sparse matrix, this can be very important when dealing with large sparse sets of equations.

If we are looking to solve BVPs and IBVPs it will be necessary to solve problems of the form,

$$\dot{\boldsymbol{y}} = \boldsymbol{D}\,\boldsymbol{y}. \tag{4}$$

Given measurements of $\dot{\boldsymbol{y}}$ we may wist to compute $\boldsymbol{y}$,

$$\boldsymbol{y} = \boldsymbol{D}^{+}\,\dot{\boldsymbol{y}} + \text{null}\left\{\boldsymbol{D}\right\}\,\boldsymbol{\alpha}. \tag{5}$$

It is known that a differential operator as a single null, i.e. the constant vector $\mathbf{1}\,\alpha$. Consequently, the matrix $\boldsymbol{D}$ should also have a single constant vector as a null space. This is not the case for the regularizing differential operator $\boldsymbol{D}_r$.

Differentiating matrices generally become degenerate if computed globally for a large number of points and a high degree, this is a fundamental property of such matrices. Comsequently, the `DOPbox` provides the function `dopDiffLocal.m` to compute a local polynomial differential approximator.

```
ls = 7;
noBfs = ls;
%
Dl = dopDiffLocal(x, ls, noBfs);
dyL = Dl * y;
%
```

```
%
fig6 = figure;
plot( x, dy - dyDop, 'r');
hold on;
plot( x, dy - dyL, 'k');
grid on;
xlabel('Support');
ylabel('Error in d y(x)');
legend('dop derivative','local derivative','location','NorthWest');
```

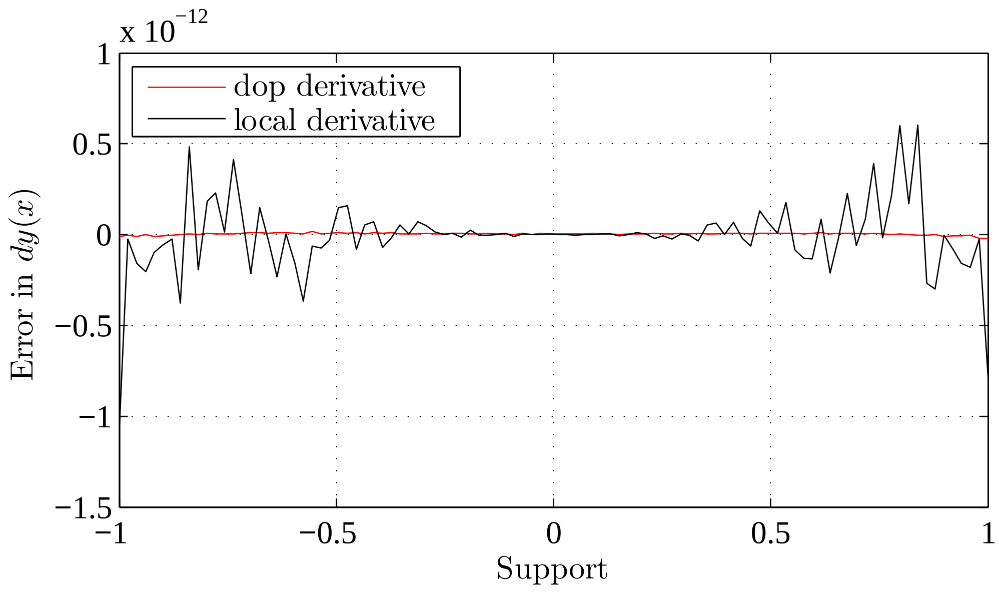


Figure 11: Comparison of the regularizing and local estimates for the derivative of $y(x)$

There are indeed larger errors in the local derivative approximation, but at a scale of 10^{-12} there are very small. The matrix has, despite being dimension 100×100 the correct null space.

A numerical test for the quality of the matrix is the use of its pseudo inverse as a numerical integrator.

```
yMat = pinv(DyMat) * dy;
yDopLocal = pinv(Dl) * dy;
%
% ensure that the curve is mean free, this eliminates the need to deal
        with
% the constant of integration.
%
yt = y - mean(y);
fig7 = figure;
plot( x, y - yMat, 'k');
hold on;
plot( x, y - yDopLocal, 'r');
grid on;
```

```
legend('Matlab Gradient','dopLocal','location','NorthWest');
```

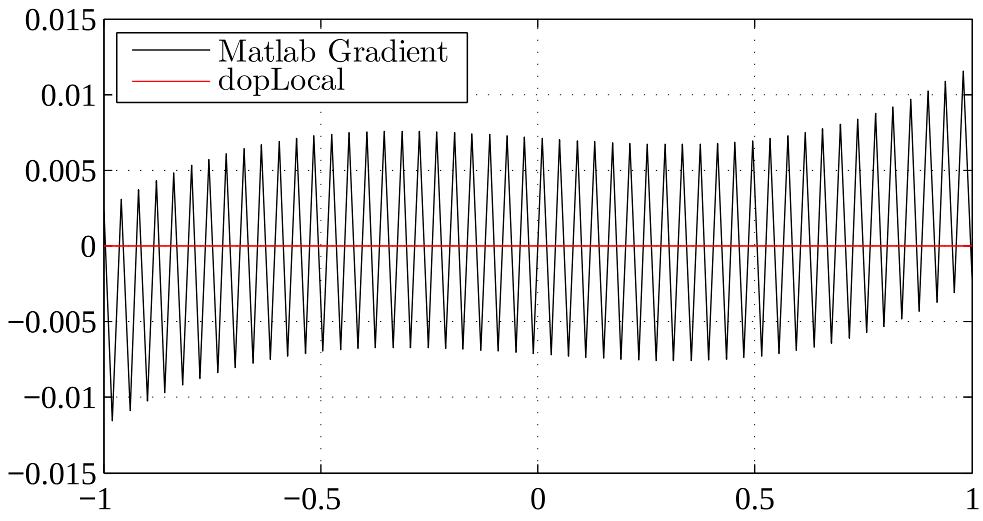


Figure 12: Error in the curve reconstruction from its derivatives using the pseudo inverse of the gradient operator in MATLAB and the differentiating matrix computed using dopLocal.

This result clearly shows that the differentiating matrix computed using `dopDiffLocal.m` is well suited for inverse problems.

2.4 Apply a set of constraints to a set of basis functions – `dopConstrain.m`

Boundary value problems, initial value problems and problems with inner constraints are characterized by a differential equation and a set of constraints which must be fulfilled by the solutions. The function `dopConstrain` enables the application of constraints to a set of basis functions.

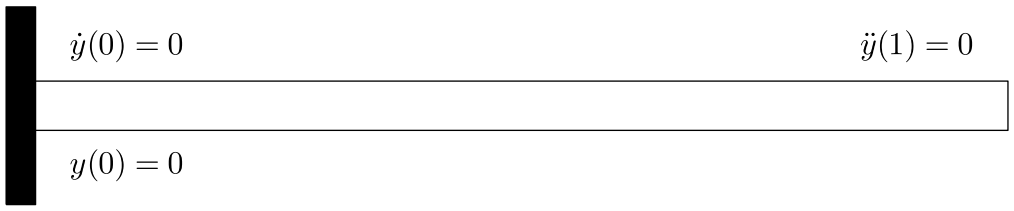


Figure 13: A simple cantilever with its associated constraints.

A simple cantilever provides a good example of a boundary value problem. The bending of the beam is described by the Euler-Bernouli equation and the following constraints need to be fulfilled, $y(0) = 0$, $\dot{y}(0) = 1$ and $\ddot{y}(1) = 0$. We now demonstrate generating a set of addmissible functions for this problem using `dopConstrain`. This is an interesting simple example since it requires both value and derivative constraints.

```
n = 100;
noBfs = 7;
%
xs = linspace(0,1,n )';
%
% And generate a set of basis functions
%
B = dop( xs, noBfs );
```

Now it is necessary to define the constraints.

```
c1 = zeros( size(xs) );
%
% the first location in c corresponds to x = 0; settin a 1 at this
        location
% required c1' * y = 0, i.e. it implements the constraint
%
c1(1) = 1;
%
% the second constraint is a derivative constraint. First we must
        compute
% the derivative matrix and extract the first row which computes the
% derivative at the first point
%
```

```
ls = 5;
D = dopDiffLocal( xs, ls, ls );
d1 = D(1,:);
c2 = d1';
% the third constraint in on the end of the beam and is on the second
% deravitive.
%
D2 = D * D;
d21 = D2(end,:);
c3 = d21';
%
% Now conatinating the constraints
%
C = [c1, c2, c3];
%
% And applying the constraints to the basis functions B
%
Bc = dopConstrain( C, B );
```

Having generated the basis functions we can now view them

```
noBcs = noBfs - rank(C);
figure;
plot( x, Bc(:,1), 'k');
hold on;
for k=2:(noBcs)
   plot( x, Bc(:,k), 'k');
end;
xlabel('Support');
ylabel('Amplitude');
grid on;
```

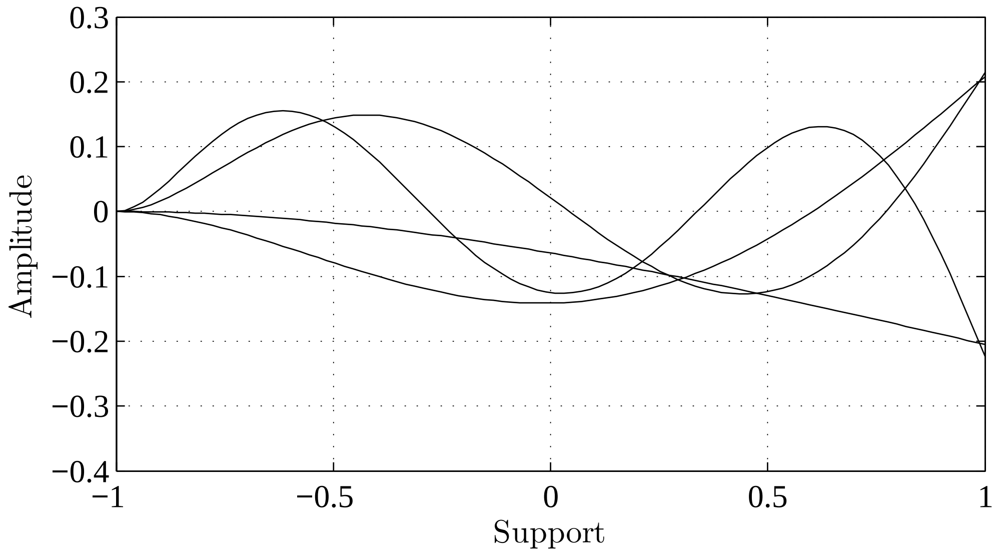


Figure 14: Constrained basis functions for the cantilever example.

The generated basis functions are also orthonormal. This makes them optimal for the propagation of Gaussian noise.

```
Gc = Bc' * Bc
GcPerp = eye( noBcs ) - Gc
```

```
Gc =
    1.0000    0.0000   -0.0000   -0.0000
    0.0000    1.0000    0.0000   -0.0000
   -0.0000    0.0000    1.0000    0.0000
   -0.0000   -0.0000    0.0000    1.0000
GcPerp =
   1.0e-15 *
         0   -0.1804    0.0069    0.2498
   -0.1804    0.6661   -0.2290    0.1457
    0.0069   -0.2290    0.6661   -0.6870
    0.2498    0.1457   -0.6870    0.4441
```

2.5 Global and local polynomial approximation – `dopApproxLocal.m`

In this section both global and local polynomial approximation is demonstrated. Global approximation referrs to applying the basis functions to the full length of the support. Given a set of noise data $\boldsymbol{y}_n$ located at the points $\boldsymbol{x}$, global approximation is performed by generating a set of basis functions $\boldsymbol{B}_x$ of the desired degree d at the nodes defined by $\boldsymbol{x}$. The approximation is performed by computing the projection onto the basis functions,

$$\boldsymbol{y}_{ag} = \boldsymbol{B}_x \, \boldsymbol{B}_x^{\mathrm{T}} \, \boldsymbol{y}_n. \tag{6}$$

This can also be looked at in a simular manner to `polyfit` and `polyval`. The coefficients of the Gram polynomial corresponding to $\boldsymbol{y}_n$ are computed as,

$$\boldsymbol{g} = \boldsymbol{B}_x^{\mathrm{T}}\, \boldsymbol{y}_n, \tag{7}$$

this corresponds to `polyfit`. The reconstruction of the signal is performed as,

$$\boldsymbol{y}_{ag} = \boldsymbol{B}_x\, \boldsymbol{g}. \tag{8}$$

corresponding to `polyval`.

There is clearly a direct relationship between the coefficients of the Gram polynomials and the geometric polynomials: this is obtained by looking at the reconstruction equation,

$$\boldsymbol{y}_{ag} = \boldsymbol{B}_x\, \boldsymbol{g} = \boldsymbol{V}\, \boldsymbol{p} \tag{9}$$

whereby, $\boldsymbol{V}$ is the Vandermonde matrix and $\boldsymbol{p}$ would be the coefficients delivered by `polyfit`. Consequently, the coefficinets $\boldsymbol{p}$ can be computed from $\boldsymbol{g}$ with the following MATLAB code,

```
p = V\(Bx * g).
```

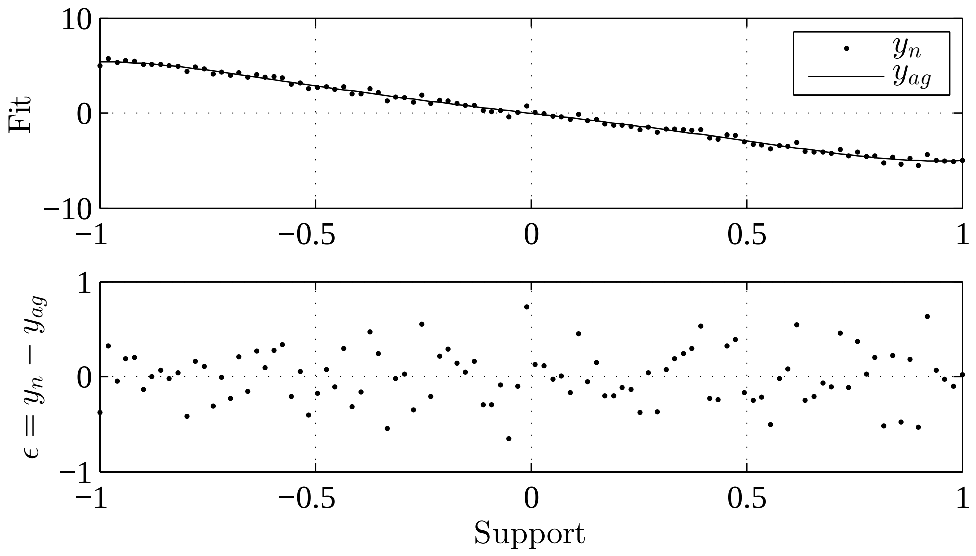
At this point it is important to note that in some applications it will be possible to compute a perfectly good approximation using the Gram polynomials; however, if the Vandermonde matrix $\boldsymbol{V}$ has become degenerate it will not be possible to compute the corresponding geometric coefficients $\boldsymbol{p}$.

Using the data from Figure 10 to demonstrate fitting

```
degree = 6;
noBfs = degree + 1;
%
% Generat the basis functions
%
Bx = dop( x, noBfs );
%
% Compute the spectrum
%
g = Bx' * yn
%
% Reconstruct, i.e., compute the approximation.
%
yag = Bx * g;
%
figure;
subplot(2,1,1);
plot( x, yn, 'k.');
hold on;
plot( x, yag, 'k');
grid on;
ylabel('Fit');
legend('y_n', 'y_{ag}','Location','NorthEast');
%
subplot(2,1,2);
plot( x, yn - yag, 'k.');
```

```
grid on;
ylabel('\epsilon = y_n - y_{ag}');
xlabel('Support');
```

```
 g =
     0.1407
   -33.4198
     0.5564
     0.7259
     0.2210
     1.2476
    -0.0029
```



However, there are many examples where a global approximation will suffer from the Runge phenomena. In such cases local polynomial approximations may provide better approximations. Consider the data shown in Figure 15 this is a data set for which Runge phenomena is relevant

```
load approxLocalData
%
figGL = figure;
plot( x, y, 'k');
hold on;
plot( x, yn, 'k.');
xlabel('Support');
ylabel('Value');
axis([-1,1,-0.1,1.1]);
grid on;
```

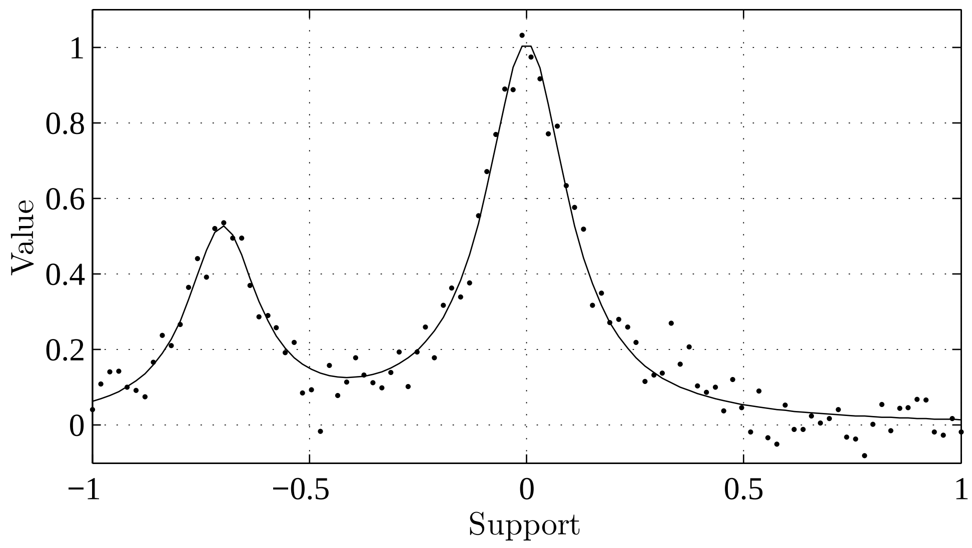


Figure 15: Test data set to compare global and local polynomial approximation.

Approximating the above data set with a degree $d = 25$ polynomial

```
degree = 25;
B = dop( x, degree + 1 );
yg = B * B' * yn;
%
figure(figGL);
plot( x, yg, 'r');
legend('y(x)','y_n','y_g');
```

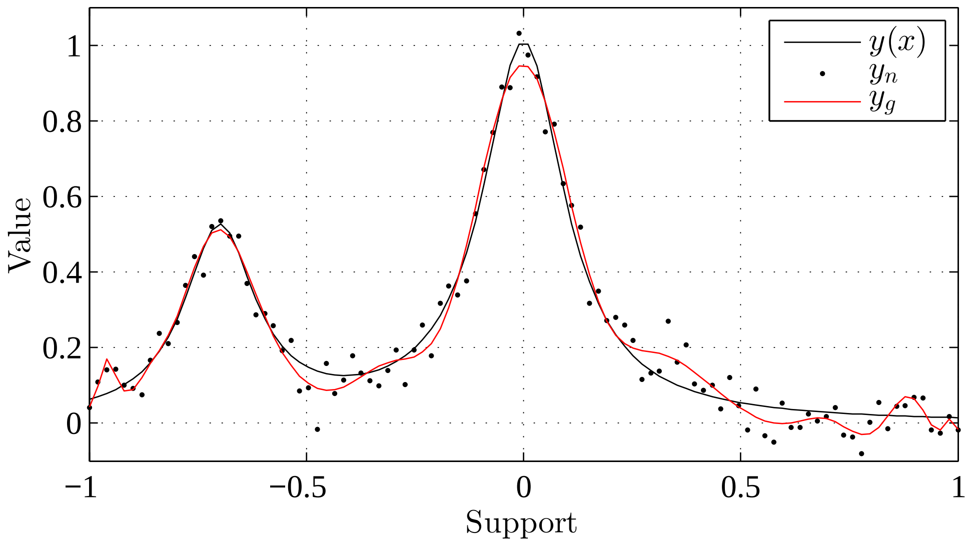


Figure 16: The degree $d = 25$ polynomial approximation exhibits a significant effect from the Runga phenomena.

Now comparing the results to a local polynomial approximation (LPA). The the LPA is characterized by the length of the support l_s, i.e. the number of points used for the local approximation and the degree d of the approximation. The function `dopApproxLocal.m` generates a matrix $\boldsymbol{S}$ which performs the LPA.

```
ls = 11;
d = 3;
%
% Generate the linear operator
%
S = dopApproxLocal( x, ls, d + 1 );
%
% Compute the approximation
%
ys = S * y;
%
figGL2 = figure;
plot( x, y, 'k');
hold on;
plot( x, yn, 'k.');
xlabel('Support');
ylabel('Value');
axis([-1,1,-0.1,1.1]);
grid on;
plot( x, ys, 'r');
legend('y(x)','y_n','y_s');
```

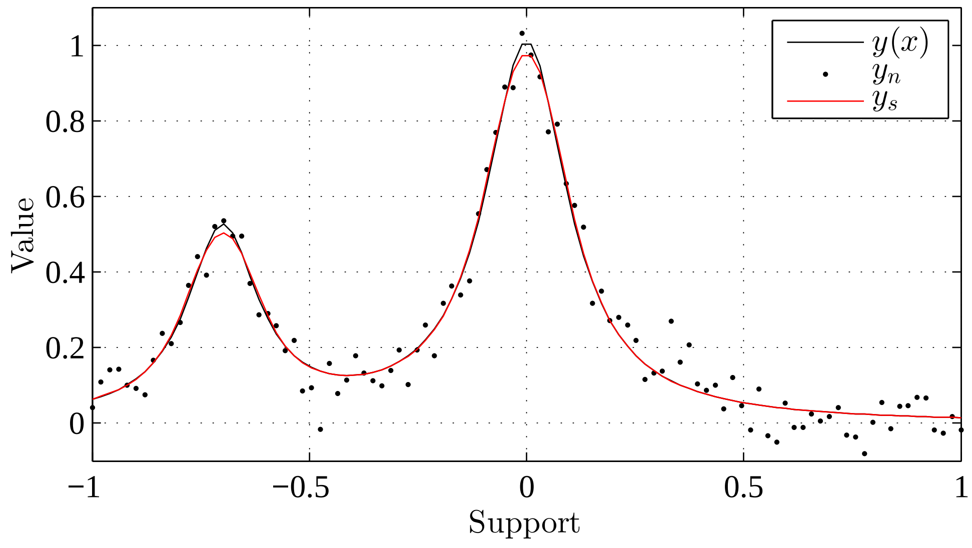


Figure 17: local polynomial approximation.

2.6 Interpolating using basis functions – `dopInterpolate.m`

The methods provided here generate basis functions for arbitrary nodes within the unit circle on the complex plane. In many cases the corresponding analytical functions are not directly available. The function `dopInterpolate` provided a measns of interpolating within such functions.

The test data set `interpolateData` cantains a set of non-uniformly spaced x values (xnu), a uniformly spaced set of nodes (x) and the function y(x) (y) evaluated at these points. the example here is to reevaluate the equations are regularly spaced nodes.

```
load interpolateData;
%
% Generate a set of basis functions for original set of nodes
%
noBfs = 5;
[B, dB, rCfs] = dop( x, noBfs );
%
% compute the spectrum of y with respect to B.
%
polyCoeffs = B' * y;
yb = B * polyCoeffs;
%
% Interpolate onto the non-uniform set of nodes
%
yi = dopInterpolate( polyCoeffs, rCfs, xnu );
%
figure;
plot( x, y, 'k');
hold on;
```

```
plot( xnu, yi, 'ro','markerFaceColor', 'w');
plot( x, y, 'k.');
grid on;
xlabel('Support');
ylabel('Amplitude');
```

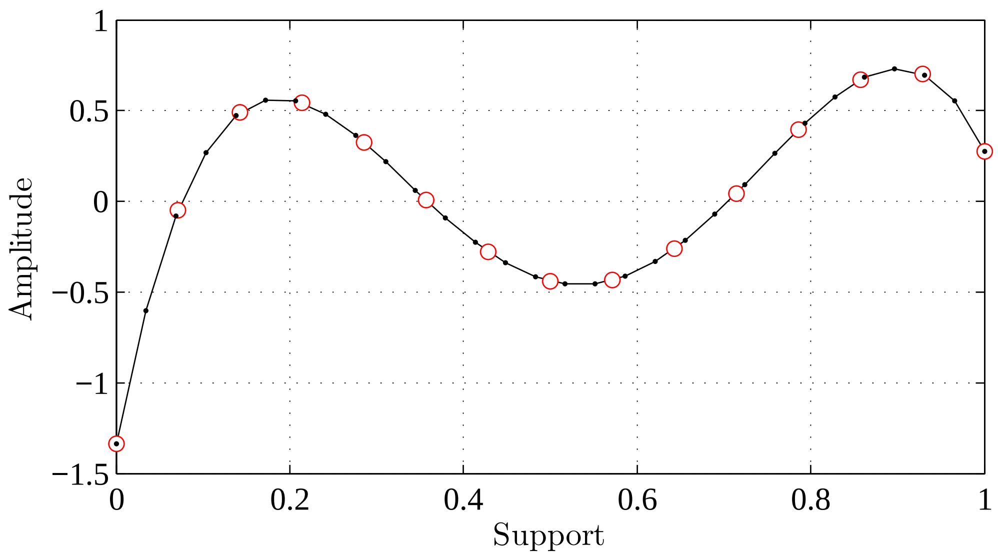


Figure 18: Example of using `dopInterpolate` to recompute a function at a new set of arbitrary nodes.

2.7 Generate a set sine basis functions – `dopSine.m`

Generate a set of sine wave basis functions and their derivatives.

```
nrS = 300;
nrBfs = 4;
[S, dS, x] = dopSine( nrS, nrBfs );
%
figS = figure;
subplot(2,1,1);
for k=1:nrBfs
   plot( x, S(:,k), 'k');
   hold on;
end;
grid on;
ylabel('sine(x)');
subplot(2,1,2);
for k=1:nrBfs
   plot( x, dS(:,k), 'k');
   hold on;
end;
```

```
grid on;
ylabel('d sine(x) / dx');
xlabel('x');
```

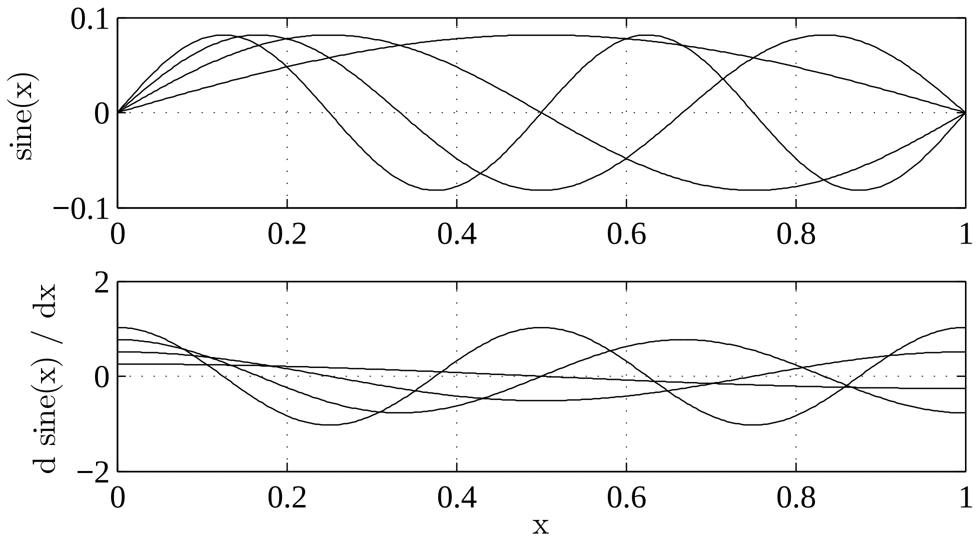


Figure 19: Sine basis functions and their derivatives.

2.8 Discrete Weighted Orthogonal Polynomials: `dopWeightBasis`

Discrete orthogonal weighted polynomials are defined on the orthogonality condition,

$$\boldsymbol{B}\,\boldsymbol{W}\,\boldsymbol{B}^{\mathrm{T}} = \boldsymbol{I} \tag{10}$$

Whereby, $\boldsymbol{W}$ is a full rank positive definite weighting matrix, it is not restricted to being a diagonal matrix.

```

%
% Example generate 10 weighted basis functions
%
nw = 100;
nrBfs = 10;
%
% generate the Gram nodes
%
xw = dopNodes( nw, 'gram');
%
% generate the Unitary basis functions
%
B = dop( xw , nrBfs );
%
```

```
% Define the weights and compute the weighted basis functions. These
% correspond to the Gegenbauer polynomials for alpha = -0.5 and scaled
        to
% be unitary in the definition B' W B = I.
%
w = 1./(1 - xw.^2);
W = diag( w );
Bw = dopWeightBasis( B, W );
%
figW = figure;
plot( xw, Bw(:,2), 'k');
hold on;
plot( xw, Bw(:,3), 'k');
plot( xw, Bw(:,4), 'k');
plot( xw, B(:,2), 'r');
plot( xw, B(:,3), 'r');
plot( xw, B(:,4), 'r');
xlabel('Support');
ylabel('Weighted polynomial value');
grid on;
```

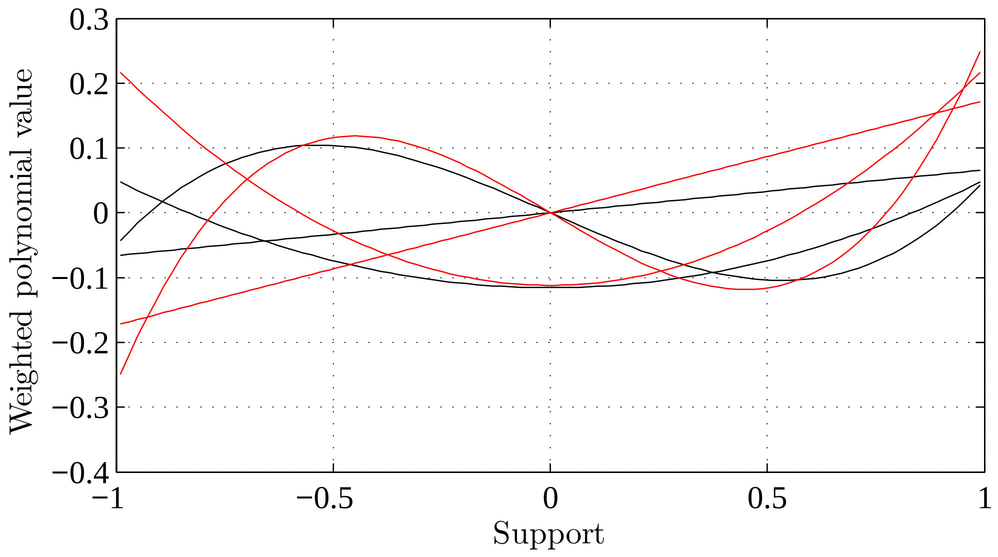


Figure 20: Discrete othogonal weighted polynomials $\boldsymbol{B}^{\mathrm{T}}\boldsymbol{W}\boldsymbol{B} = \boldsymbol{I}$. The wighting used here is $w = \frac{1}{1-x^2}$. Weighted polynomials in black and unweighted in red.

2.9 Weighted regression with weighted basis functions

The weighted basis functions presented in the previous section can used for a simple implementation of a weighted regression, e.g., a covariance weighted regression.

The cost function for weigthed regression is,

$$E = \left(\boldsymbol{y} - \boldsymbol{B}_w\,\boldsymbol{\alpha}^{\mathrm{T}}\right)\,\boldsymbol{W}\,(\boldsymbol{y} - \boldsymbol{B}_w\,\boldsymbol{\alpha})\,. \tag{11}$$

In the case of covariance weighted regression $\boldsymbol{W} = \boldsymbol{\Lambda}^{-1}$ whereby Λ^{-1} is the symmetric positive definite covariance matrix. Now expanding the above equation and differentiating with respect to $\boldsymbol{\alpha}$ yields,

$$\frac{\mathrm{d}E}{\mathrm{d}\boldsymbol{\alpha}} = -2\,\boldsymbol{B}_w^{\mathrm{T}}\,\boldsymbol{W}\,\boldsymbol{y} + 2\,\boldsymbol{B}_w^{\mathrm{T}}\,\boldsymbol{W}\,\boldsymbol{B}_w\,\boldsymbol{\alpha}. \tag{12}$$

Evaluating the derivative equal to the zero vector, yields the so-called normal equations for weighted regression,

$$\boldsymbol{B}_w^{\mathrm{T}}\,\boldsymbol{W}\,\boldsymbol{B}_w\,\boldsymbol{\alpha} = \boldsymbol{B}_w^{\mathrm{T}}\,\boldsymbol{W}\,\boldsymbol{y}. \tag{13}$$

If the weighted basis functions are generated such that $\boldsymbol{B}_w^{\mathrm{T}}\,\boldsymbol{W}\,\boldsymbol{B}_w = \boldsymbol{I}$ as shown in the previous section. Then the normal equations simplify to:

$$\boldsymbol{\alpha} = \boldsymbol{B}_w^{\mathrm{T}}\,\boldsymbol{W}\,\boldsymbol{y}. \tag{14}$$

Computing in this manner avoide the necessity to calculate matrix inverses at the time of fitting. This greatly simplifies the computation of weighted regression.

2.10 `dopFit` and `dopVal`

These are two wrapper functions to simplify the use of the DOP library for fitting and evaluating discrete orthogonal polynomials. They are the discrete orthogonal polynomial equivalent of `polyfit` and `polyval`. However, the discrete orthogonal polynomials can perform fitting up to degree of $D = 1000$ without serious numerical errors.

The function `dopVit` also returen the covariance matrix for the polynomial coefficients and `dopVal` returen the covariance matrix for the fitted $\boldsymbol{y}_f$ values.

A Taylor expansion for a function is equivalent to computing a polynomial approximation. The discrete orthogonal polynomials enable the computation of a Gram expansion, i.e. a polynomial expansion, of very high degree. This may be advantegous, e.g., when solving differential equations which are known to have no analytical solutions such as the Mathieu differential equation.

2.10.1 Fitting a high degree polynomials

In this first test a polynomial of degree $d = 100$ is fit to a data set. Fitting with this high degree is not possible with the standard MATLAB function `polyfit`.

Load a test data set:

```
load dopFitData;
%
% Perform the fit and evaluate the polynomial
%
d = 100;
[g, Lg, S] = dopFit( x, y, d );
yg = dopVal( g, S );
%
```

```
% Compute the residual
%
r = y - yg;
%
% Plot the fit
%
fitFig1 = figure;
plot( x, y, 'k.');
hold on;
plot( x, yg, 'r');
ylabel('Value');
grid on;
xlabel('x');
```

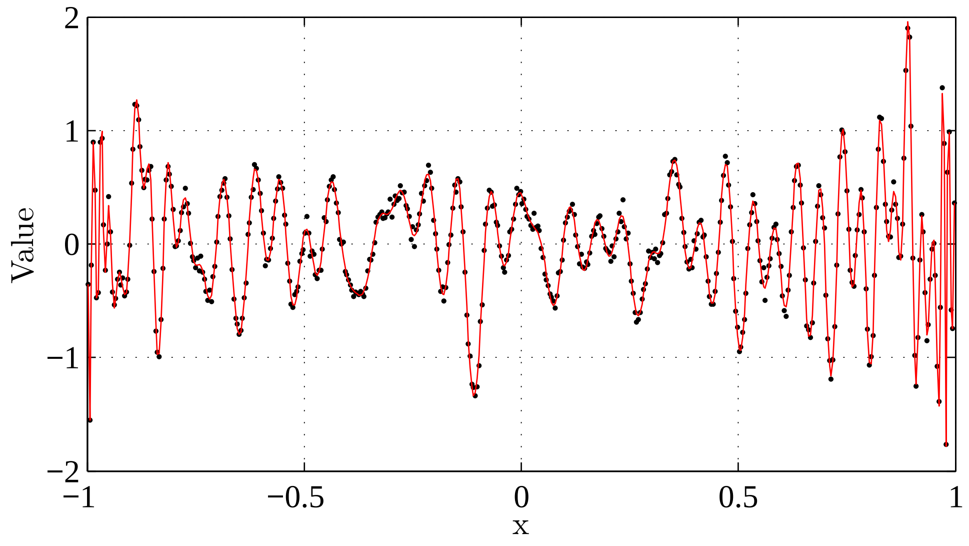


Figure 21: Example of a discrete orthogonal polynomial fit with degree $d = 100$.

Plot the residual

```
fitFig2 = figure;
plot( x, r, 'k.');
ylabel('Residual');
grid on;
```

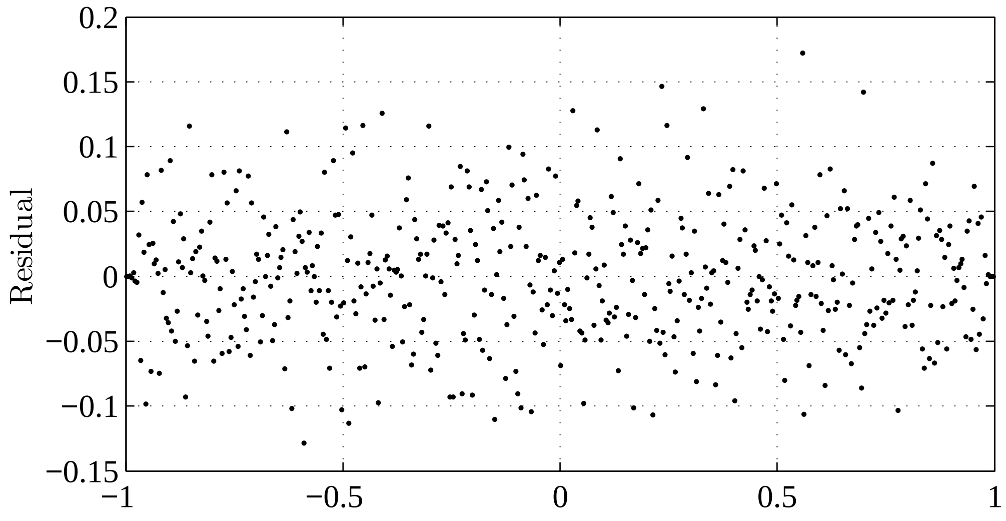


Figure 22: Residual of the discrete orthogonal polynomial fit with degree $d = 100$.

Compute the histogram of the residual

```
nrBins = 15;
fitFig3 = figure;
hist( r, nrBins );
xlabel('Residual');
ylabel('Frequency');
```

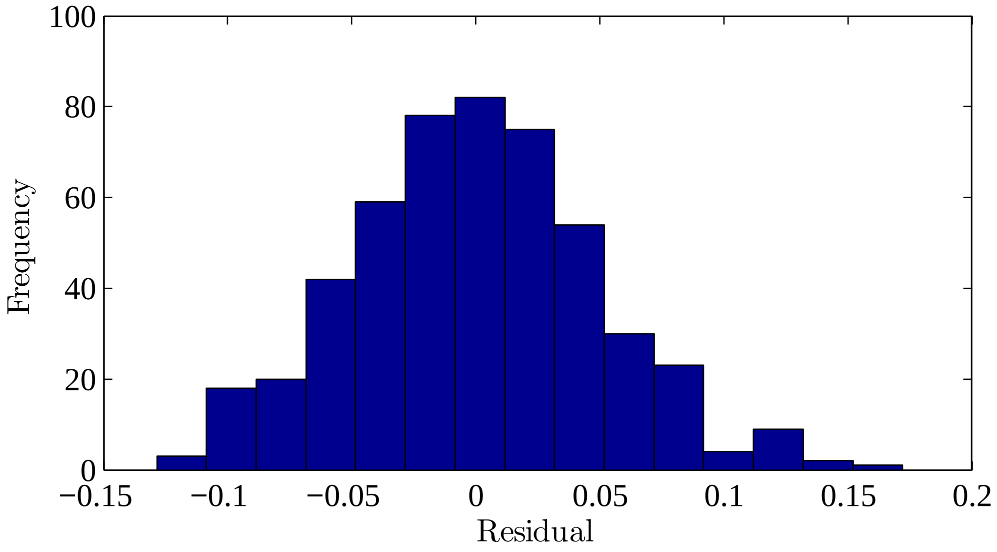


Figure 23: Histogram of the residual of the discrete orthogonal polynomial fit with degree $d = 100$. This demonstrated the Gaussian nature of the residual, i.e. the fit has not produced systematic errors.

2.10.2 High degree polynomial expansion

This section demonstrates the application of `dopFit` to compute a high degree polynomial expansion for a function. The cosine finction is chosen here because its Taylor expansion is known; furthermore, `polyfit` fails to model this data correctly.

Generate the test function

```
nrCycles = 15;
nrPts = 300;
%
x = dopNodes( nrPts, 'GramEnds');
%
phi = linspace( 0, 2*pi*nrCycles, nrPts )';
y = cos(phi);
%
% compute the fit
%
d = 70;
[g, Lg, S] = dopFit( x, y, d );
%
% Note the structure S also delivers the values of y fit. These value
        are
% required internally to do the covariance computations.
%
expFig = figure;
plot(x, y, 'k.');
hold on;
plot( x, S.yg, 'r');
xlabel('x');
ylabel('Value');
```

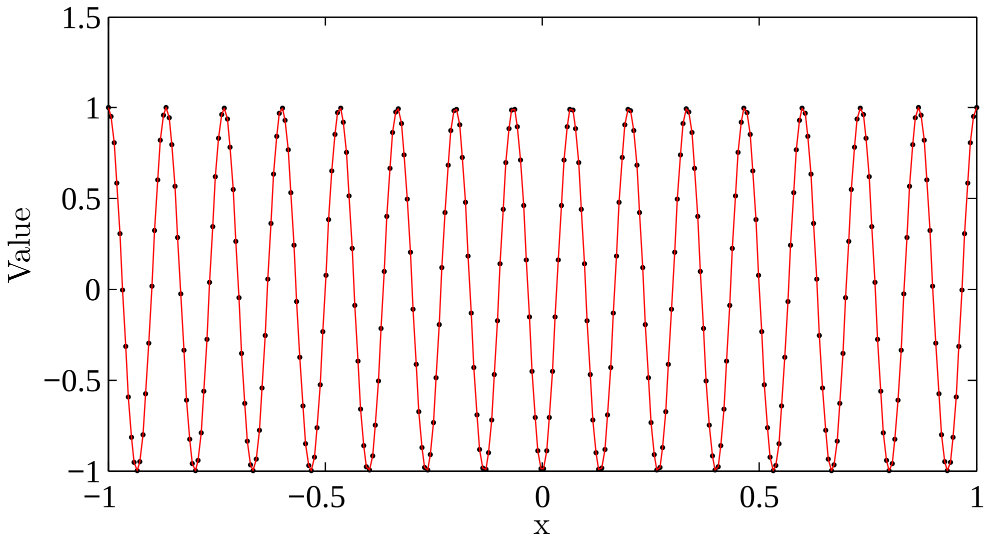


Figure 24: A discrete orthogonal polynomial expansion of degree $d = 70$ for a cosine. This demonstrated the possability of computing a polynomial expandions for functions which require a high degree.

References

[1] P. O'Leary and M. Harker. An algebraic framework for discrete basis functions in computer vision. In *2008* 6*th* *ICVGIP*, pages 150–157, Bhubaneswar, India, 2008. IEEE.

[2] P. O'Leary and M. Harker. Discrete polynomial moments and savitzky-golay smoothing. In *Waset Special Journal*, volume 72, pages 439–443, 2010.

[3] Paul O'Leary and Matthew Harker. A framework for the evaluation of inclinometer data in the measurement of structures. *IEEE T. Instrumentation and Measurement*, 61(5):1237–1251, 2012.