Testing the accuracy of the algorithms in the package

This live script is used to test the accuracy of the various codes in this software package. We begin by selecting the size of the rules that we will be testing. This size is used throughout the script.

```
n = 1000;
```

The following sections will test the accuracy of the codes against another known reference method. Once computed we will visualize the differences by plotting the scaled absolute errors $\frac{|v_1 - v_2|}{\epsilon}$ where v_1 will be the nodes from one method and v_2 will be the nodes from the reference method of computation. We will also plot the absolute relative errors of the weights $\left|\frac{w_1 - w_2}{w_2}\right|$ although these will be presented as a semi-log plot in y to make the differences more visible.

Comparison of various techniques on Chebyshev rules of the first kind

We begin our validation by comparing several techniques on Chebyshev rules of the first kind since the nodes and weights of these are known in terms of simple explicit formulas. The first step is to generate the 'truth'.

```
TrueNodes = cos((.5+(n-1:-1:0)')/n*pi);
TrueWeights = pi/n;
```

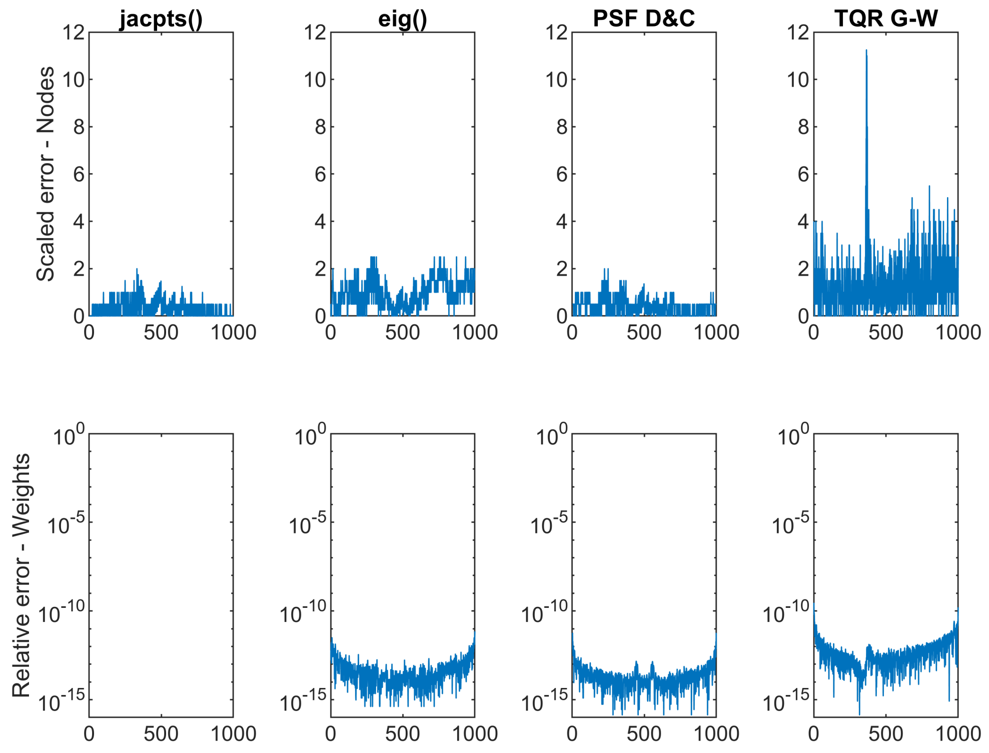
Next we construct the recurrence relation for the OPS and use that to test four different approaches.

```
[alpha, beta] = JacobiRecurrence(n,-.5,-.5);
% Nodes and weights using Matlab's eig().
[Vm,MatlabNodes] = eig(mxt(alpha,sqrt(beta(2:end))),'vector');
MatlabWeights = beta(1)*(Vm(1,:).*Vm(1,:))';
% Nodes and weights using Chebfun's jacpts().
[ChebfunNodes,ChebfunWeights] = jacpts(n,-.5,-.5); ChebfunWeights = ChebfunWeights';
% Nodes and weights using the TQR Golub-Welsch code.
[tqrnodes,tqrweights] = GaussRule(alpha,beta,'tqrGW');
% Nodes and weights using the Partial Spectral Factorization D&C code.
[DandCnodes,DandCweights] = GaussRule(alpha,beta,'dandcPSF');
```

Now we plot the absolute error using a semilogy() plot. Note: When the difference is zero that region of the plot will be empty.

```
tiledlayout(2,4);
ax1 = nexttile;
plot(abs(ChebfunNodes-TrueNodes)/eps); title('jacpts()'); ylabel('Scaled error -
Nodes')
ax2 = nexttile;
plot(abs(MatlabNodes-TrueNodes)/eps); title('eig()');
ax3 = nexttile;
plot(abs(DandCnodes-TrueNodes)/eps); title('PSF D&C');
ax4 = nexttile;
plot(abs(tqrnodes-TrueNodes)/eps); title('TQR G-W');
```

```
ax5 = nexttile;
semilogy(abs(ChebfunWeights-TrueWeights)/TrueWeights);  ylabel('Relative error - 
Weights')
ax6 = nexttile;
semilogy(abs(MatlabWeights-TrueWeights)/TrueWeights);
ax7 = nexttile;
semilogy(abs(DandCweights-TrueWeights)/TrueWeights);
ax8 = nexttile;
semilogy(abs(tqrweights-TrueWeights)/TrueWeights);
linkaxes([ax1 ax2 ax3 ax4],'xy'); linkaxes([ax5 ax6 ax7 ax8],'xy');
```



It is very important to note that although the Matlab eig() computation is used as ground truth in many papers on computing quadrature rules, it is far from that. In this case it is actually less accurate than both the jacpts() and Partial Spectral Factorization by D&C approaches. Furthermore, the perfect accuracy of the jacpts() code on computing weights in this section results from the fact that it traps certain special cases (this is one of them) and uses other methods of computing (e.g. this case is recognized as having equal weights and the jacpts() code enforces that).

Validating the GaussRule code on Gauss-Jacobi Rules

In this section we compute Gauss-Jacobi rules in three different ways: Using both Partial Spectral Factorization codes (D&C and TQR), and using the Chebfun built-in jacpts(). We take the jacpts() computation as a baseline and then compute the difference between the other two approaches and the baseline. NOTE:This is not a true test of accuracy as all three methods are approximations. We begin by choosing our Jacobi parameters (a,b) and generating the recurrence coefficients.

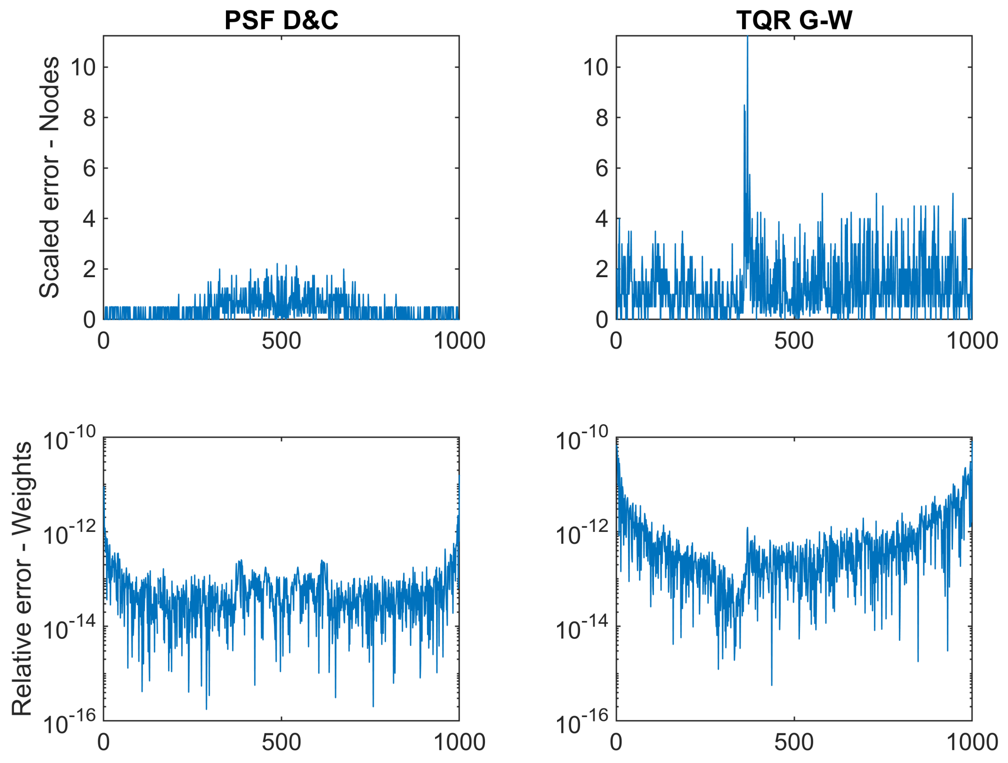
```
a=0;b=0;  % The parameter choice a=0;b=0; yields Gauss-Legendre rules but the user
is free to try others.
% Generate the recurrence coefficients.
[alpha, beta] = JacobiRecurrence(n,a,b);
```

Next, compute the nodes and weights using chebfun's jacpts() and then by using both methods available in GaussRule(). And put the results in a common format (nodes and weights as column vectors.)

```
[ChebfunNodes,ChebfunWeights] = jacpts(n,a,b); ChebfunWeights = ChebfunWeights'; %
Gauss-Jacobi rule using Chebfun's jacpts().
[tqrnodes,tqrweights] = GaussRule(alpha,beta,'tqrGW'); % GaussRule() using the TQR
Golub-Welsch code.
[DandCnodes,DandCweights] = GaussRule(alpha,beta,'dandcPSF'); % GaussRule() using
the Partial Spectral Factorization D&C code.
```

Plot the scaled differences:

```
clf; tiledlayout(2,2);
ax1 = nexttile;
plot(abs(DandCnodes-ChebfunNodes)/eps); title('PSF D&C'); ylabel('Scaled error -
Nodes')
ax2 = nexttile;
plot(abs(tqrnodes-ChebfunNodes)/eps); title('TQR G-W');
ax3 = nexttile;
semilogy(abs(DandCweights-ChebfunWeights)./abs(ChebfunWeights)); ylabel('Relative
error - Weights')
ax4 = nexttile;
semilogy(abs(tqrweights-ChebfunWeights)./abs(ChebfunWeights));
linkaxes([ax1 ax2],'xy'); linkaxes([ax3 ax4],'xy');
```



Validating the GaussPlusGaussLobatto code

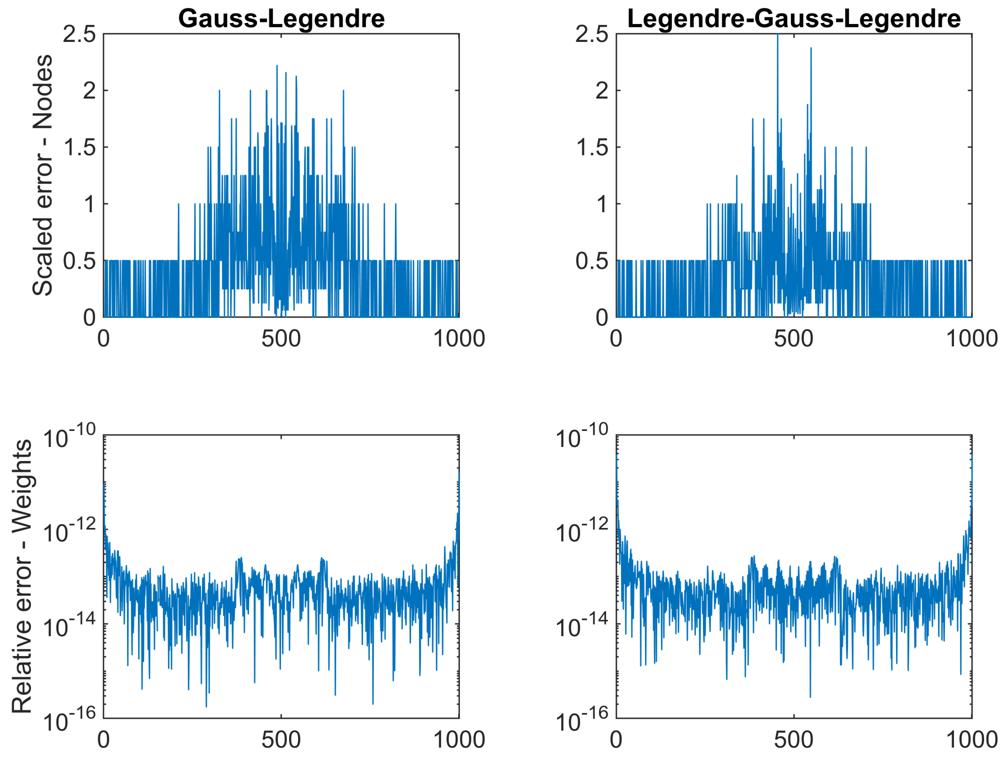
This section demonstrates the accuracy of the GaussPlusGaussLobatto code by using it to compute an n-point Gauss-Legendre plus an (n+1)-point Gauss-Lobatto rule and comparing the results to those given by the Chebfun built-ins legpts(n) and lobpts(n+1).

```
% Use Chebfun to generate the rules.
[LG_nodes,LG_weights] = legpts(n); LG_weights = LG_weights'; % Gauss-Legendre rule
using Chebfun's legpts().
[LGL_nodes,LGL_weights] = lobpts(n+1); LGL_weights = LGL_weights'; % Legendre-Gauss-
Lobatto rule using Chebfun's lobpts().
% Generate the Legendre recurrence coefficients.
[alpha, beta] = JacobiRecurrence(n,0,0);
% Generate both rules using GaussPlusGaussLobatto().
[DLGnodes,DLGweights,DLGLnodes,DLGLweights] =
GaussPlusGaussLobatto(alpha,beta,-1,1);
```

Generate the plots

```
clf; tiledlayout(2,2); ax1 = nexttile;
plot(abs(DLGnodes-LG_nodes)/eps); ylabel('Scaled error - Nodes'); title('Gauss-
Legendre');
ax2 = nexttile;
plot(abs(DLGLnodes-LGL_nodes)/eps); title('Legendre-Gauss-Legendre');
ax3 = nexttile;
```

```
semilogy(abs(DLGweights-LG_weights)./abs(LG_weights)); ylabel('Relative error -
Weights');
ax4 = nexttile;
semilogy(abs(DLGLweights-LGL_weights)./abs(LGL_weights));
linkaxes([ax1 ax2],'xy'); linkaxes([ax3 ax4],'xy');
```



Validating the GaussPlusGaussRadau code

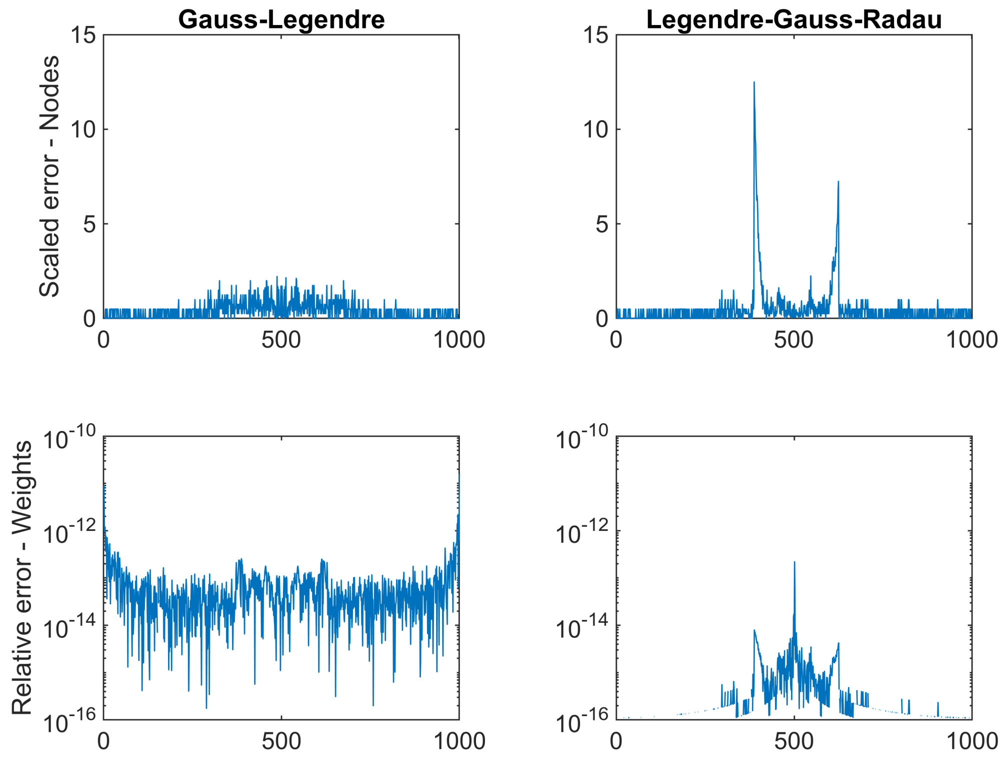
This section demonstrates the accuracy of the GaussPlusGaussRadau code by using it to compute an n-point Gauss-Legendre plus an (n+1)-point Gauss-Radau rule and comparing the results to those given by the Chebfun built-ins legpts(n) and radaupts(n+1).

```
[LG_nodes,LG_weights] = legpts(n); LG_weights = LG_weights'; % Gauss-Legendre rule
using Chebfun's legpts().
[LGR_nodes,LGR_weights] = radaupts(n+1); LGR_weights = LGR_weights'; % Legendre-
Gauss-Radau rule using Chebfun's radaupts().
% Generate the Legendre recurrence coefficients.
[alpha, beta] = JacobiRecurrence(n+1,0,0);
% Generate both rules using GaussPlusGaussRadau().
[DLGnodes,DLGweights,DLGRnodes,DLGRweights] = GaussPlusGaussRadau(alpha,beta,-1);
```

Generate the plots

```
clf; tiledlayout(2,2); ax1 = nexttile;
plot(abs(LG_nodes-DLGnodes)/eps); ylabel('Scaled error - Nodes'); title('Gauss-
Legendre');
```

```
ax2 = nexttile;
plot(abs(LGR_nodes-DLGRnodes)/eps); title('Legendre-Gauss-Radau');
ax3 = nexttile;
semilogy(abs(LG_weights-DLGweights)./abs(LG_weights)); ylabel('Relative error -
Weights');
ax4 = nexttile;
semilogy(abs(LGR_nodes-DLGRnodes)./abs(LGR_nodes));
linkaxes([ax1 ax2],'xy'); linkaxes([ax3 ax4],'xy');
```


Gauss-Legendre
Legendre-Gauss-Radau
Scaled error - Nodes
Relative error - Weights
10^{-10}
10^{-12}
10^{-14}
10^{-16}