2 Fundamental Calculations

Larry C. Young, tildentechnologies.com

Table of Contents

2. Fundamental Calculations

The purpose of this chapter is to describe the methods used to calculate some of the basic quantities needed to implement Methods of Weighted Residuals (MWR), especially orthogonal collocation or pseudospectral methods. In a collocation method, the selection of the collocation points is critically important. Once the points are selected, the method is basically specified, since the other parameters in the approximation can be calculated from the points. The orthogonal collocation or pseudospectral method differs from an ordinary collocation method by using collocation points that are the roots of orthogonal polynomials. The orthogonal polynomial roots all fall in the interior of the domain, so boundary points are added to facilitate the approximation of boundary conditions.

Why use the roots of orthogonal polynomials? In most cases they are selected because they form the basis of highly accurate numerical integration or quadrature methods. The accurate quadrature formulas produce a collocation method which closely approximates the accurate but more cumbersome integrated MWR, e.g Galerkin or Moments.

The trial solution in an MWR is sometimes a series of orthogonal polynomials. With orthogonal polynomial basis functions the trial solution looks like:

$$y(x) \approx \sum_{k=0}^{n+1} P_k(x)\,\hat{a}_k \tag{2.1}$$

where P_k are orthogonal polynomials, usually either Chebyshev or Legendre polynomials. This approach is called a *modal* method, since the unknown coefficients, $\hat{a}_k$, are analogous to the modes in a Fourier series.

Some applications use a modal basis, but as explained in Chapter 1, it is far more common to solve for the *nodal* values of the variables, so the trial solution is:

$$y(x) \approx \sum_{i=0}^{n+1} \ell_i(x)\, y(x_i) \tag{2.2}$$

where $\ell_i(x)$ are the Lagrange interpolating polynomials:

$$\ell_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{n+1} \frac{(x - x_j)}{(x_i - x_j)} = \frac{\hat{p}_n(x)}{(x - x_i)\hat{p}_n'(x_i)}$$

and:

$$\hat{p}_n(x) = \prod_{j=0}^{n+1}(x - x_j) \quad \text{and} \quad \hat{p}_n'(x_i) = \left.\frac{d\hat{p}_n(x)}{dx}\right|_{x_i} = \prod_{\substack{j=0 \\ j\neq i}}^{n+1}(x_i - x_j)$$

The boundary points are x_0 and x_{n+1}. The left definition of $\ell_i(x)$ above is the familiar one and the right definition is called the fundamental form [Szegö (1975)].

With a Lagrange polynomial basis, one solves for the values at the collocation points or nodes, so the method is called a *nodal* method. The nodal approach gives a more intuitive, finite-difference-like method. Some articles state that a modal basis is used, but through a series of operations the problem is formulated in terms of nodal values. There are simple linear transforms that can be used to convert from one representation to the other. These transformations are discussed in Sections 2.6 and 2.7. However, it is more clear to develop the approximation with one consistent basis. This monograph will employ a nodal basis almost exclusively. A few simple examples use a modal approach in order to give a flavor for the differences between the two.

In order to apply MWR solution methods, we must be able to calculate various derivatives of these trial functions. For the integrated MWR described in Chapter 1, i.e. Galerkin and Moments methods, integration is obviously important. Collocation does not require integration per se and some developments of these methods pay little attention to it. However, integration also plays a key role in efficient collocation methods, so the application of all MWR requires techniques for integrating and differentiating quantities involving the trial functions, whether nodal or modal. The purpose of this chapter is to develop the relationships and methods needed to calculate the coefficients needed to solve problems with MWR.

The terms $1/\hat{p}_n'(x_i)$ in Eq. (2.2) are the so called barycentric weights. The integration and differentiation relationships are formulated to depend on these same quantities, so once the polynomial roots or collocation points are known, the other quantities can be calculated easily.

Since the nodes in Eq. (2.2) consist of the roots of an orthogonal polynomial together with the endpoints, the orthogonal polynomial is related by:

$$\hat{p}_n(x) = (x - x_0)(x - x_{n+1})\, p_n(x) \tag{2.3}$$

where $p_n(x)$ in Eq. (2.2) is the monic form of the orthogonal polynomial, i.e. with leading coefficient (of x^n) of unity. Here we follow the convention that the monic form of an orthogonal polynomial is denoted by a lowercase p and the conventional form is in uppercase. A hat (^) designates the inclusion of the endpoints.

Differentiation of Eq. (2.2) is straight forward, but integration is more of a problem. The integration formulas needed for nonsymmetric problems are of the form:

$$\int_0^1 f(x)dx \cong \sum_{i=0}^{n+1} f(x_i) \int_0^1 \ell_i(x)\,dx = \sum_{i=0}^{n+1} W_i f(x_i) \quad \text{or}$$
$$W_i = \int_0^1 \ell_i(x)\,dx \tag{2.4}$$

For symmetric problems, the x in Eq. (2.2) is replaced by x^2 and since the symmetry condition is automatically met at x = 0, only the endpoint at x = 1 is included. In this case the points are numbered from 1 to n+1 instead of 0 to n+1. The numerical integration formulas needed for symmetric problems are of the form:

$$\int_0^1 f(x^2)x^\gamma dx = \frac{1}{2}\int_0^1 f(\xi)\xi^\kappa d\xi \cong \frac{1}{2}\sum_{i=1}^{n+1} f(\xi_i) \int_0^1 \ell_i(\xi)\,\xi^\kappa d\xi = \sum_{i=1}^{n+1} W_i f(\xi_i) \quad \text{or}$$
$$W_i = \frac{1}{2}\int_0^1 \ell_i(\xi)\xi^\kappa d\xi \tag{2.5}$$

where $\xi = x^2$, γ = 0,1,2 and $\kappa = (\gamma - 1)/2$, so κ = -½, 0, +½ for planar, cylindrical or spherical geometry. These are called interpolatory quadrature formulas, because the approximate integration can be derived by integration of the interpolant of the integrand.

Kopal (1955) has a particularly clear and understandable description of numerical integration methods. Eq. (2.4) has a total of n+2 weights. In an equally spaced Newton-Cotes method, these free parameters can fit the coefficients of an n+1 degree polynomial to achieve exact integration. After one is familiar with Newton-Cotes quadrature, it seems almost unbelievable that it is possible to achieve almost twice the accuracy with a similar formula, i.e. 2n+1 degrees. Take a moment to consider this possibility. Since Eq. (2.4) includes the endpoints by definition, there are 2n+2 free parameters, n+2 weights and n base points. We could use brute force and set up a system of equations to solve for the weights and basepoints to produce exact integrals through 2n+1 degrees. We would find that a solution is possible and except for the endpoints, all the base points are within the interval $0 < x_i < 1$. The result would be Lobatto quadrature which is described in Section 2.3. Lobatto quadrature is a close cousin of Gaussian quadrature. Gaussian quadrature achieves the highest accuracy, 2n-1 degrees, for a given number of base points, so it minimizes the number of function evaluations, $f(x_i)$. It differs from Lobatto quadrature, because endpoint weights are not utilized, i.e. $W_0 = W_{n+1} = 0$. Clenshaw-Curtis (1960) quadrature uses Chebyshev points as the quadrature base points. The points are

not optimally located for these integrals, so like Newton-Cotes only an n+1 degree polynomial can be integrated exactly.

This chapter on basic calculations is divided into several parts. To apply orthogonal collocation or pseudospectral methods, one first needs the polynomial roots or quadrature base points, $\boldsymbol{x}$. Section 2.1 describes the properties of the orthogonal polynomials for which the roots are sought, while Section 2.2 discusses two methods for calculating their roots. Section 2.3 describes quadrature methods and calculation of the quadrature weights, i.e. $\boldsymbol{W}$, in Eqs. (2.4) and (2.5). To solve differential equations with MWR, one must differentiate the basis functions. Section 2.4 covers the differentiation of the orthogonal polynomials, Eq. (2.1), needed to implement a modal method. The differentiation matrices for first and second derivatives and related quantities in a nodal approximation are described in Section 2.5. Although this monograph uses nodal formulations, the relationship to other representations is described. Section 2.6 discusses the transformation between modal and nodal bases, i.e. Eq. (2.1) vs (2.2), while Section 2.7 describes transforms from nodal to monomial bases. Software to calculate all the quantities described in this chapter is discussed in Section 2.8. Several example calculations using supplied software and methods of this chapter are described in Section 2.9.

Knowledge of the details of how these various quantities are calculated is not essential for one to apply these methods. Formulas for the various quantities could be just written down, but some background information is included here for those interested in a deeper understanding of the method. We do not delve into the proofs and development of various formulas which are readily available elsewhere, but give the results that are needed for our purposes and the applicable reference for details. For general references to the subject of orthogonal polynomials and quadrature the reader is directed to Hildebrand (1987) and Krylov (1975).

2.1 Jacobi Polynomials

Jacobi polynomials are a family of orthogonal polynomials which include all the polynomials of interest for solving nonperiodic problems on a finite interval. Orthogonal polynomials and quadrature formulas are conventionally based on the interval [-1,1], while the normal orthogonal collocation convention uses the more convenient interval [0,1]. On the interval [0,1] the polynomials are called *shifted* polynomials. The interval [-1,1] is used here for this fundamental development. Once the fundamental properties are established, the corresponding properties for the shifted polynomials are given. Any interval can be used with a suitable transformation.

Jacobi polynomials are orthogonal with respect to a weight function with parameters α and β. For endpoints a and b they meet the orthogonality condition:

$$\int_a^b (b-x)^\alpha (x-a)^\beta P_n^{(\alpha,\beta)}(x) P_m^{(\alpha,\beta)}(x) dx = \zeta_n^{(\alpha,\beta)} \delta_{nm} \tag{2.6}$$

where by convention for a = -1 and b = 1:

$$\zeta_n^{(\alpha,\beta)} = \frac{2^{\alpha+\beta+1}\,\Gamma(n+\alpha+1)\,\Gamma(n+\beta+1)}{(2n+\alpha+\beta+1)\,\Gamma(n+\alpha+\beta+1)\,n!}$$

The requirement of orthogonality establishes the polynomials only within a multiplicative constant. Several conventions could be used to complete the specification. If orthonormality is required the polynomial would be scaled so that ζ = 1. Alternatively, monic polynomials could be specified. The convention above is established from the endpoint condition given below in Eq. (2.10). The Legendre polynomials correspond to $\alpha = \beta = 0$ and Chebyshev polynomials of the 2nd kind to $\alpha = \beta = ½$. These polynomials are called *ultraspherical* for the common case when $\alpha = \beta$. This special case can also be represented by a Gegenbauer polynomial.

Orthogonal polynomials are closely tied to the theory of accurate quadrature methods. For Eq. (2.4), Gaussian quadrature gives the highest accuracy for a given number of quadrature base points. Consider a more general integration of the form:

$$\int_a^b f(x)\omega(x)dx = \sum_{i=1}^{m} W_i^* f(x_i) \tag{2.7}$$

where: $\omega(x) = (b-x)^\alpha (x-a)^\beta$. The optimal quadrature is Gauss-Jacobi quadrature, where the base points are the roots of the Jacobi polynomials $P_m^{(\alpha,\beta)}$. It can be proven that all the roots lie within the interval, $a < x < b$. Gauss-Jacobi quadrature will produce exact integrals when $f(x)$ is a polynomial of degree $2m$-1.

Krylov (1975) shows that if some of the base points are prescribed and the others are determined for optimal accuracy; the weighting function in the orthogonal polynomial must be zero at the specified points. Lobatto quadrature includes both endpoints, so the weight function is made to be zero at both ends by taking $\alpha = \beta = 1$. Lobatto quadrature interior base points are the roots of the corresponding Jacobi polynomials. As we shall see these points also correspond to the extrema for the Legendre polynomials. For symmetric problems if a weight is included at the endpoint x = 1 it is correctly called a Radau quadrature; however, these formulas correspond to the right half of a Lobatto formula, so we will call it Lobatto in all cases to simplify the terminology. For symmetric problems β must correspond to κ in Eq. (2.5).

The Gauss-Jacobi weights in Eq. (2.7) can be used to specify weights for most of the quadrature formulas of interest. Gaussian quadrature is the Gauss-Jacobi quadrature for the special case where $\alpha = \beta = 0$. An expression for the Lobatto quadrature weights can be derived from the Gauss-Jacobi weights for $\alpha = \beta = 1$. Since the interpolating polynomials include the endpoints [0,1], following Eq. (2.4) the interior quadrature weights are related by:

$$W_i = \int_0^1 \ell_i(x)\,dx = \int_0^1 \left[\frac{x(1-x)}{x_i(1-x_i)} \prod_{\substack{j=1 \\ j\neq i}}^{n} \frac{(x-x_j)}{(x_i-x_j)}\right] dx = \frac{W_i^*}{x_i(1-x_i)} \tag{2.8}$$

The boundary weights are easily calculated from $\sum W_i = 1$ or as described in Section 2.3. Compared to Gaussian quadrature with the same number of interior points, this formula achieves two additional degrees of accuracy due to the $x(1 - x)$ term. However, the endpoint weights are nonzero here, while they are zero or not used in Gaussian quadrature. The relative accuracy of these two methods agree with the heuristic arguments made in conjunction with Eq. (2.4)

Taking these factors into account for symmetric and nonsymmetric problems of various geometry, Table 2.1 summarizes the specific Jacobi polynomials of interest for quadrature. Nonsymmetric problems in cylindrical and spherical geometry are not considered because such problems make sense only if the other periodic angular coordinates are included and we do not consider periodic problems here.

Table 2.1 Weight Exponents for Jacobi Polynomials

	Planar	Cylindrical	Spherical
Nonsymmetric, Gauss	$\alpha = 0, \beta = 0$	n.a.	n.a.
Nonsymmetric, Lobatto	$\alpha = 1, \beta = 1$	n.a.	n.a.
Symmetric, Gauss	$\alpha = 0, \beta =$ -½	$\alpha = 0, \beta = 0$	$\alpha = 0, \beta =$ ½
Symmetric, Lobatto	$\alpha = 1, \beta =$ -½	$\alpha = 1, \beta = 0$	$\alpha = 1, \beta =$ ½

Note that the Chebyshev weight factors, $\alpha = \beta =$ -½ for the 1st kind and $\alpha = \beta =$ +½ for the 2nd kind, do not appear in Table 2.1. Their roots give optimal quadratures only for integrands involving radicals $\sqrt{x(1-x)}$ or $1/\sqrt{x(1-x)}$. One can define MWR which includes radicals in the weighting of the residual function (see Eq. (1.4)), but such methods are not considered here. Chebyshev polynomials are considered in this monograph primarily because they are a popular choice due to some of their computational advantages.

The Jacobi polynomials and their properties can be determined directly from Eq. (2.6). The first two Jacobi polynomials on [-1,1] are:

$$P_0^{(\alpha,\beta)} = 1, \qquad P_1^{(\alpha,\beta)} = \frac{1}{2}(\alpha + \beta + 2)x + \frac{1}{2}(\alpha - \beta) \tag{2.9}$$

The polynomials are designated with superscript (α,β) when needed to avoid ambiguity. For Legendre or generic polynomials or when the meaning is obvious no superscript is used. When $\alpha = \beta$ the polynomials are alternately odd and even or symmetric and antisymmetric about $x = 0$.

The endpoints for the conventional form of the polynomials are:

$$P_n^{(\alpha,\beta)}(1) = \frac{\Gamma(n + \alpha + 1)}{n!\,\Gamma(\alpha + 1)} \quad \text{and}$$
$$P_n^{(\alpha,\beta)}(-1) = (-1)^n \frac{\Gamma(n + \beta + 1)}{n!\,\Gamma(\beta + 1)} \tag{2.10}$$

For Legendre polynomials, the values at x = 1 are unity and are alternating ±1 at x = -1. For the Jacobi $\alpha = \beta$ = 1 polynomials they are n + 1 at x = 1 and alternating ± (n + 1) at x = -1. The endpoint values in Eq. (2.10) establish the convention by which the formula for ζ is determined in Eq. (2.6) for the conventional form of Jacobi polynomials.

The polynomials can be expanded as linear combination of monomials like Eq. (2.9), but for the higher order polynomials the coefficients become large with alternating signs. If used for calculations in that form roundoff errors soon become important. There is a better way to evaluate the polynomials.

The orthogonality condition, Eq. (2.6), can be used to develop a simple recurrence relationship. For this development, the superscript (α,β) will be dropped for convenience. The recurrence formula is:

$$p_{n+1} = (x - \hat{\alpha}_n)p_n - \hat{\beta}_n p_{n-1} \tag{2.11}$$

First define $p_0 = 1,\ p_{-1} = 0,$ and $\hat{\beta}_0 = \int_{-1}^{1} \omega(x)\,dx$, then the other parameters are:

$$\hat{\alpha}_n = \frac{\int_{-1}^{1} p_n^2\, x\, \omega(x)dx}{\int_{-1}^{1} p_n^2\, \omega(x)dx} = \frac{\beta^2 - \alpha^2}{(2n+\alpha+\beta)(2n+\alpha+\beta+2)} \quad \text{and}$$

$$\hat{\beta}_n = \frac{\int_{-1}^{1} p_n^2\, \omega(x)dx}{\int_{-1}^{1} p_{n-1}^2\, \omega(x)dx} = \frac{4n(n+\alpha)(n+\beta)(n+\alpha+\beta)}{(2n+\alpha+\beta)^2[(2n+\alpha+\beta)^2 - 1]}$$

Eq. (2.11) can be derived directly from the orthogonality condition, Eq. (2.6), and from the fact that any polynomial can be expressed as a linear combination of orthogonal polynomials.

Eq. (2.11) is the recurrence relationship for the monic polynomials. If the leading coefficients are defined by:

$$P_n = \rho_n p_n \tag{2.12}$$

The leading coefficient can be determined from:

$$\zeta_n = \int_{-1}^{1} \rho_n^2\, p_n^2\, \omega(x)dx = \rho_n^2 \prod_{k=0}^{n} \hat{\beta}_k \tag{2.13}$$

For the convention given in Eq. (2.6), the leading coefficients are:

$$\rho_n = \frac{\Gamma(2n+\alpha+\beta+1)}{2^n n!\,\Gamma(n+\alpha+\beta+1)} \tag{2.14}$$

A recurrence relationship for the conventional form of the Jacobi polynomials can easily be determined from the expressions above:

$$P_{n+1} = \left[x\left(\frac{\rho_{n+1}}{\rho_n}\right) - \left(\hat{\alpha}_n \frac{\rho_{n+1}}{\rho_n}\right)\right] P_n - \left(\hat{\beta}_n \frac{\rho_{n+1}}{\rho_{n-1}}\right) P_{n-1} \tag{2.15}$$

These equations are for polynomials on the interval [-1,1]. They can be converted to the interval [0,1], which is more convenient and has become the standard for orthogonal collocation applications. These are called *shifted* polynomials. Using a tilde (~) to indicate the corresponding values for shifted polynomials, the parameters are:

$$\begin{aligned}\tilde{\zeta}_n^{(\alpha,\beta)} &= \zeta_n^{(\alpha,\beta)}/(2^{\alpha+\beta+1})\\ \tilde{\rho}_n^{(\alpha,\beta)} &= (2^n)\rho_n^{(\alpha,\beta)}\\ \tilde{\alpha}_n^{(\alpha,\beta)} &= \tfrac{1}{2}\left(1+\hat{\alpha}_n^{(\alpha,\beta)}\right)\\ \tilde{\beta}_n^{(\alpha,\beta)} &= \tfrac{1}{4}\hat{\beta}_n^{(\alpha,\beta)}\end{aligned} \tag{2.16}$$

As an example, consider the Legendre polynomials. The recurrence relation and leading coefficients for the interval [-1,1] are:

$$\begin{aligned}\hat{\alpha}_n &= 0\\ \hat{\beta}_n &= \frac{n^2}{4n^2-1}\\ \rho_n &= \frac{(2\mathrm{n})!}{2^n(n!)^2}\\ \zeta_n &= \frac{2}{2n+1}\end{aligned} \tag{2.17}$$

The recurrence relationship for the monic form of the Legendre polynomials is then:

$$p_{n+1} = x\,p_n - \left(\frac{n^2}{4n^2-1}\right)p_{n-1} \tag{2.18}$$

While the recurrence relationship for the conventional form of the Legendre polynomials is given by:

$$P_{n+1} = \left(\frac{2n+1}{n+1}\right)xP_n - \left(\frac{n}{n+1}\right)P_{n-1} \tag{2.19}$$

The first few Legendre polynomials are:

$$P_0 = 1,\ P_1 = x,\ \ P_2 = \frac{1}{2}(3x^2-1),\ \ P_3 = \frac{1}{2}(5x^3-3x) \tag{2.20}$$

Fig. 2.1 shows some examples of 5th and 6th order Jacobi polynomials. Note that the polynomials tend to look like sine or cosine curves in the middle of the interval, but are compressed near the boundaries. Note also that the endpoints of the polynomials appear to be in agreement with Eq. (2.10) (although the scale cuts off the extreme portions). $P_6^{(1,0)}$ is the only one shown which is not symmetric (or antisymmetric) about the center point. It also follows the Legendre polynomial near x = 0 and the $P_6^{(1,1)}$ polynomial

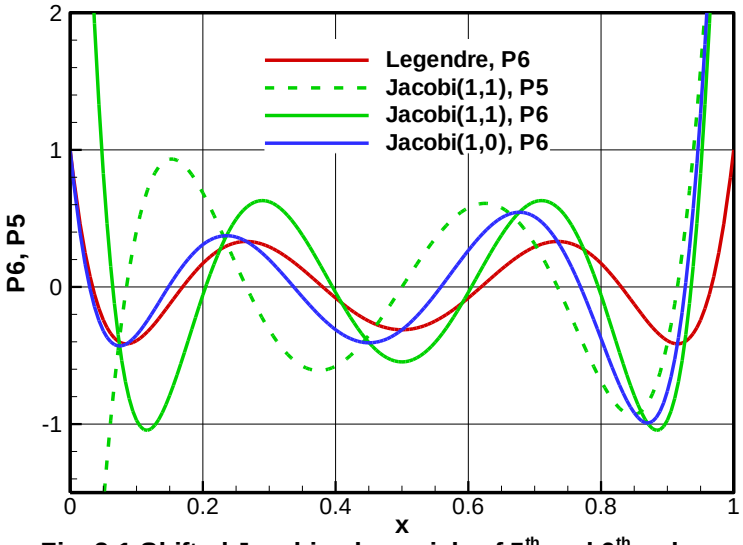

Fig. 2.1 Shifted Jacobi polynomials of 5th and 6th order

near x = 1. Finally, note that the roots of $P_5^{(1,1)}$ coincide with the extrema of the Legendre polynomial. This last feature will be explained shortly.

2.2 Orthogonal Polynomial Roots

One advantage of using Chebyshev points is that their roots can be directly calculated. The roots of Chebyshev polynomials of the 2nd kind on the interval [-1,1] are:

$$x_i = -\cos\left(\frac{i\pi}{n+1}\right) \tag{2.21}$$

for i = 0,…,n + 1.

For other polynomials, the roots cannot be found as easily. There are two basic methods which can be used to determine the roots of the other Jacobi polynomials. The most popular method reformulates the problem as an eigenvalue problem for which the roots are the eigenvalues. The other is an iterative method such as the Newton-Raphson method. Before discussing these methods, we first describe a shortcut which is applicable to either method.

Many of the cases in Table 2.1 have roots that are symmetric about the center point. A symmetric point distribution occurs for the ultraspherical case, i.e. $\alpha = \beta$. A simple way to exploit this symmetry in an iterative method is to solve for only half the points and copy them to the others. Table 2.1 hints at a much better method for exploiting this symmetry. The points for a symmetric problem in planar geometry are the right half of those for a nonsymmetric problem. For example, for Lobatto points, use α = 1 and β = -½ and take the square root of the roots that are determined on the interval [0,1]. These values correspond to the positive roots for the full polynomial on [-1,1]. This method reduces the number of roots by 2 and the degree of the polynomial by 2, so the calculations are reduced by approximately a factor of 4. It works quite well for an even number of points.

Although not as straight forward as for even n, a similar method can be applied for odd n [Krylov (1975) pp 117-121]. When $\alpha = \beta$ and the polynomial is odd, there are an odd number of roots and one root is in the center, i.e. x = 0. Division by x creates a symmetric polynomial, which can be treated like the case when n is even. Suppose we associate two symmetric polynomials:

$$S_n(x) = \frac{P_{2n+1}\left(\sqrt{x}\right)}{\sqrt{x}} \text{ or } S_n(x)\,S_m(x)\sqrt{x} = \frac{P_{2n+1}\left(\sqrt{x}\right)P_{2m+1}\left(\sqrt{x}\right)}{\sqrt{x}} \tag{2.22}$$

The relationship of their orthogonality is as follows:

$$\int_0^1 S_n(x)\,S_m(x)\sqrt{x}\,dx = \int_0^1 P_{2n+1}\left(\sqrt{x}\right)P_{2m+1}\left(\sqrt{x}\right)\frac{dx}{\sqrt{x}} = 2\int_0^1 P_{2n+1}(\xi)\,P_{2m+1}(\xi)\,d\xi = \int_{-1}^1 P_{2n+1}(\xi)\,P_{2m+1}(\xi)\,d\xi = 0 \text{ for } m \neq n \tag{2.23}$$

where $\xi = \sqrt{x}$. So, the positive roots of P_{2n+1} (on [-1,1]) are the square root of the roots of S_n (on [0,1] with β = +½). The other roots are the corresponding negative ones and the one at x = 0. If the roots of S_n found on [-1,1] are $\boldsymbol{x_s}$, the positive roots of P_{2n+1} are $\boldsymbol{x_p} = \sqrt{(1+\boldsymbol{x_s})/2}$. Then small errors are related by $\boldsymbol{\epsilon_p} = \boldsymbol{\epsilon_s}/(4\boldsymbol{x_p})$, where $\boldsymbol{\epsilon_p}$ and $\boldsymbol{\epsilon_s}$ are the errors in the roots of P_{2n+1} and S_n, respectively. This example, Eq. (2.23), is for Gauss points, $\alpha = \beta = 0$, but the technique works the same for Lobatto points, $\alpha = \beta = 1$, by solving for the roots with $\alpha = 1$ and β = +½.

We refer to the procedure just described, which solves for half the roots as the *shortcut* method. It can be applied for odd or even n by using β = ±½. It can be applied to either of the methods described below, the eigenvalue method or iterative method, and will reduce the calculation effort by approximately a factor of 4 in each case.

A method described by Golub and Welch (1969) notes that the problem can be cast in the form of an eigenvalue problem for a symmetric tridiagonal matrix, called the Jacobi matrix. The matrix is constructed from the recurrence coefficients as follows:

$$\begin{bmatrix} \hat{\alpha}_0 & \sqrt{\hat{\beta}_1} & & & 0 \\ \sqrt{\hat{\beta}_1} & \hat{\alpha}_1 & \sqrt{\hat{\beta}_2} & & \\ & \sqrt{\hat{\beta}_2} & \ddots & \ddots & \\ & & \ddots & \ddots & \ddots \\ 0 & & & \ddots & \ddots \end{bmatrix} \tag{2.24}$$

The recurrence relationship arranged in this manner gives the polynomials in *orthonormal* form. The orthogonal polynomial is the characteristic polynomial of this matrix, so its eigenvalues are the roots of the polynomial. Given the recurrence coefficients and a routine for calculating eigenvalues, the roots can be found with only a few lines of code (see code box below). The quadrature weights can also be found from the eigenproblem, since they bear a simple relationship to the eigenvectors of the Jacobi matrix. However, the code below

Matlab Function for Quadrature and Differentiation

```
function [x,w,A] = OCnonsymGLR(n,meth)
   % code for nonsymmetric orthogonal collocation applications on 0 < x < 1
   % n - interior points
   % meth = 1,2,3,4 for Gauss, Lobatto, Radau (right), Radau (left)
   % x - collocation points
   % w - quadrature weights
   % A - 1st derivative
   na = [1 0 0 1];  nb = [1 0 1 0];  nt = n + 2;
   a = 1.0 - na(meth);  b = 1.0 - nb(meth);
   ab = r_jacobi(n,a,b);  ab(2:n,2) = sqrt(ab(2:n,2));
   T = diag(ab(2:n,2),-1) + diag(ab(:,1)) + diag(ab(2:n,2),+1);
   x = eig(T);  x=sort(x);  x=0.5*(x+1.0);  x = [0.0;x;1.0];
   xdif = x-x'+eye(nt);     dpx = prod(xdif,2);
   w = (x.^nb(meth)).*((1.0 .- x).^na(meth))./(dpx.*dpx); w = w/sum(w);
   A = dpx./(dpx'.*xdif);  A(1:nt+1:nt*nt) = 1.0 - sum(A,2);
 end
```

calculates the weights using the method described in Section 2.3. It also calculates the differentiation matrix using the same parameters as described in Section 2.5. The code relies on Gautschi's (2005) OPQ `r_jacobi` function to obtain the recurrence coefficients.

The other method for determining the roots is a standard iterative method, such as the Newton-Raphson method. For the iterative solution of a problem like this, one must make certain all roots are found and they are in order. Iterative algorithms often use a coarse scan and isolate step before switching to a Newton-Raphson method for refinement to the final value. The methods described here can start with a Newton-Raphson method and will converge rapidly and without fail. Our goal here is to develop a method which is reasonably efficient and valid for up to 100 points or so.

If s indicates the iteration number, each iteration of a normal Newton-Raphson method requires calculation of the polynomial value and its derivative. These quantities are used to find an improved estimate of the k^{th} root as follows:

$$x_k^{s+1} = x_k^s - p_n(x_k^s)/p_n'(x_k^s) \tag{2.25}$$

where p_n and its derivative are calculated with the recurrence relationship, Eq. (2.11). The derivative of the polynomial could be calculated using Eq. (2.45) described in Section 2.4, but unless many iterations are required, it is probably more efficient to use direct differentiation of the recurrence relationship:

$$p_{n+1}' = p_n + (x - \hat{\alpha}_n)p_n' - \hat{\beta}_n p_{n-1}' \tag{2.26}$$

Villadsen and Michelsen (1978) devised a particularly clever iterative method which is based on a Newton-Raphson iteration with suppression of roots found. Suppose that roots 0 through k - 1 have been found and we want to calculate root k with an algorithm modified to suppress the roots already found. The lower order polynomial which includes only the remaining roots is not explicitly calculated, but the benefits can be obtained nevertheless. The polynomial with suppressed roots is:

$$r_k(x) = \frac{p_n(x)}{q_k(x)} \tag{2.27}$$

where $q_k(x) = \prod_{i=0}^{k-1}(x - x_i)$ with $q_0 = 1$ and its derivative is $q_k'(x) = q_k(x)\sum_{i=0}^{k-1} 1/(x - x_i)$. Eq. (2.25) is then modified to:

$$x_k^{s+1} = x_k^s - \frac{r_k(x_k^s)}{r_k'(x_k^s)} = x_k^s - \frac{p_n}{p_n'\left[1 - \frac{p_n q_k'}{p_n' q_k}\right]} \tag{2.28}$$

Fig. 2.2 illustrates this method for finding the roots of the monic Jacobi polynomial, $p_5^{(1,1)}$. The figure depicts an iteration seeking the 3rd root after the first two have been found. The curves are the full polynomial and the one with the roots suppressed, i.e. $r_2(x)$ in Eq. (2.27). The straight line is the Newton-Raphson linear estimate for the next root. This method has two advantages. First, the iterations should converge faster, because as the iterations proceed, the degree of the polynomial is continually reduced until the last root is on a straight line,

requiring no iteration. Second, the radius of convergence is larger and more predictable as shown by the heavy line in Fig. 2.2, so convergence can be assured. Any point to the left of the next root is within the radius of convergence. The method will converge without fail by starting near the left boundary and by always selecting an initial guess which is less than the value of the next root, i.e. $x_k^0 < x_k$. Alternatively, we could start at the right boundary and work back to the left.

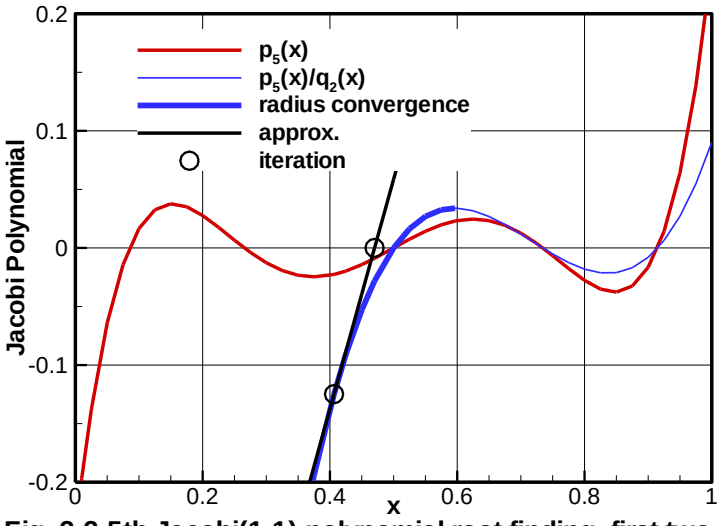

Fig. 2.2 5th Jacobi(1,1) polynomial root finding, first two roots suppressed

In practice, there is a better method for selecting initial guesses. Excellent methods are available for estimating the roots of Jacobi polynomials. With an accurate estimate convergence is assured and fewer iterations are required. One formula which works well is due to Gatteschi [Gautschi and Giordano (2008)]:

$$x_k = -\cos(\theta_k + \delta\theta_k) + O(n^{-4}) \tag{2.29}$$

where

$$\theta_k = \pi\left(2k + \beta - \tfrac{1}{2}\right)/\sigma$$

$$\delta\theta_k = \left[\left(\tfrac{1}{4} - \beta^2\right)\cot\left(\frac{\theta_k}{2}\right) - \left(\tfrac{1}{4} - \alpha^2\right)\tan\left(\frac{\theta_k}{2}\right)\right]/\sigma^2$$

$$\sigma = 2n + \alpha + \beta + 1$$

This equation is exact for all combinations of Chebyshev points, α = ±½ and β = ±½. Its stated applicability is for $|\alpha| \le ½$ and $|\beta| \le ½$. The estimated roots are usually accurate to at least four digits even for relatively small values of n. The accuracy is better for $\alpha = \beta = 1$ (Lobatto) than for $\alpha = \beta = 0$ (Gauss), even though the Lobatto case is outside the stated range of applicability. Tests have shown that when Eq. (2.29) is used for initialization of Lobatto points, only two Newton-Raphson iterations are needed to achieve 16 digits of accuracy, i.e. to the limit of normal double precision (8 byte) calculations. Other methods for initial estimates frequently require 5 or 6 iterations for equivalent accuracy. For Gauss points the results are not as good, with 3 iterations normally required for equivalent accuracy. These conditions seem to apply for a wide range of n, because as n increases and the problem becomes more difficult, the initial estimates are more accurate.

With Eq. (2.29), the one or two points near the boundaries often require an additional iteration relative to those in the interior. The additional iterations are because the estimates are not as good near the boundaries. These polynomials are trig-like away from the boundaries, but are more like Bessell functions near the boundaries. Recall from Eq. (2.10) and Fig. 2.1 that the behavior near x = 1 is controlled by α, while β dominates the behavior near x = -1. Reversing α

and β creates a mirror image of the polynomial. Eq. (2.29) is accurate near boundaries with α or β of ±½, but its accuracy is not as good near other boundaries. A more accurate approximation for the roots near the boundary, x = 1, is also due to Gatteschi [Gautschi and Giordano (2008)]. The approximation has a form similar to Eq. (2.29):

$$x_k = \cos\left(\tilde{\theta}_k + \delta\tilde{\theta}_k\right) + O\left(J_{\alpha,k}^5\, n^{-7}\right) \tag{2.30}$$

where:

$$\tilde{\theta}_k = \frac{J_{\alpha,k}}{\nu}$$

$$\delta\tilde{\theta}_k = -\tilde{\theta}_k\left[\frac{4-\alpha^2-15\beta^2}{720\nu^4}\left(\frac{J_{\alpha,k}^2}{2}+\alpha^2-1\right)\right]$$

$$\nu = \tfrac{1}{2}\sqrt{\sigma^2+(1-\alpha^2-3\beta^2)/3}$$

and $J_{\alpha,k}$ are the zeroes of the Bessel functions of order α. $J_{\alpha,k}^5$ increases rapidly with k, so the accuracy deteriorates rapidly as the roots move away from the boundary. Eq. (2.30) can be used near x = -1 by reversing α and β. We will refer to the formula above, Eq. (2.30), as the boundary correlation and the other, Eq. (2.29), as the interior correlation. Using this boundary correlation is more than overkill for our stated objective, but it was investigated out of curiosity.

Fig. 2.3 shows the error in the positive roots for n = 6 and 14 for Gauss ($\alpha = \beta$ = 0) and Lobatto ($\alpha = \beta$ = 1) points. These calculations used the shortcut method, so only 3 and 7 roots were calculated with β = -½. After the square root conversion required for the shortcut method (see below Eq. (2.23)), the correlations give the same root estimates for either the shortcut or full method. As explained above, the transformed error is given by $\boldsymbol{\epsilon_p} = \boldsymbol{\epsilon_s}/(4\boldsymbol{x_p})$, where $\boldsymbol{\epsilon_p}$ and $\boldsymbol{x_p}$ refer to the error for the full polynomial and the value for its root on [-1,1] and $\boldsymbol{\epsilon_s}$ refers to the error for the root of the reduced polynomial on [-1,1]. The error plotted in Fig. 2.3 are for the full polynomial, i.e. $\boldsymbol{\epsilon_p}$.

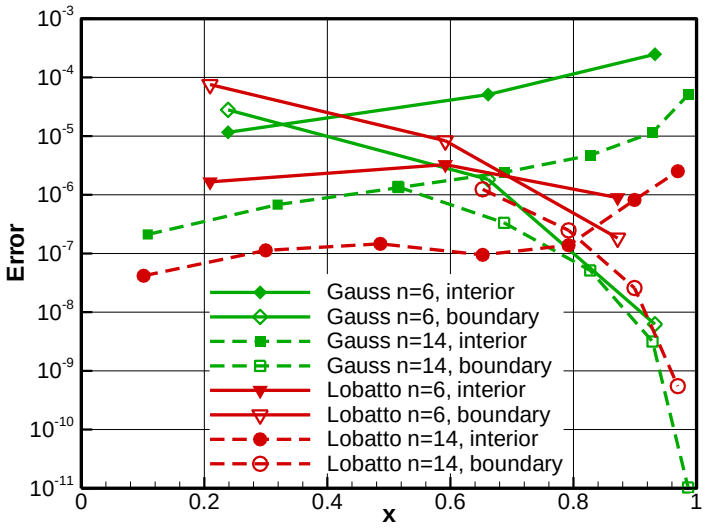

Fig. 2.3 Error in estimated roots by correlations

Fig. 2.3 shows when comparing the boundary correlation, Eq. (2.30), to the interior one, Eq. (2.29), the boundary correlation is favored when α = 0 more so than when α = 1. For α = 0 the boundary correlation is better for 2 of 3 and 5 of 7 points when n = 6 and 14, respectively, whereas 1 of 3 and 2 of 7 points are more accurate for α = 1. With the interior correlation, Eq. (2.29), the error tends to be nearly constant away from the boundary, but turns upward for the points nearest the boundary, by almost two orders of magnitude for n = 14. One could use the values of α, β and n to decide how many roots to approximate with the boundary correlation.

Obviously, complex relationships determine the best choice, and an accurate determination is beyond our needs. Our objective is to reduce the error for the points nearest the boundary, so they do not require more than two iterations. In the end, we used Eq. (2.30) for the two points nearest the boundary when $n > 2$. The boundary correlation is also used for points near x = -1 when β = 0 or 1. With this approach, the convergence is usually slowest for either the second or the third root from the boundary, but it is not appreciably slower than surrounding points. Uniform convergence is needed for the vector code described below. One could consider other estimates, some specific to the Gauss case [Hale and Townsend (2013)].

The idea of root suppression is a good one, but with accurate initial estimates it is not needed. When the current estimate is close to convergence, the bracketed term in the denominator of Eq. (2.28) goes to unity, so root suppression has no effect. A disadvantage of root suppression is the roots are found one at a time. Newer computer hardware has again made vectorization of code important. Writing code for vectorization is especially important for interpreted languages like Matlab even when vector hardware is not available. For these reasons, we have elected not to use root suppression and instead solve for all roots together. The code should vectorize with appropriate hardware and compilers.

The text box below shows code which should vectorize. It relies on the `r_jacobi` function as does the eigenvalue code above. It also uses a function `RootEstimate` (not shown) to provide initial estimates as described above. The iteration loop requires three statements, one to get the change in the value according to Eq. (2.25), one to update the values, and a test for convergence. Each statement in the iteration loop, including those in `Pcalc`, operates on all n roots at once. This code should be more efficient for Matlab, since interpreted languages are

Matlab Function for Jacobi Roots

```
function [x] = Jacobi_Xroot(n,a,b)
   xtol = 1000.0*eps;   MaxNR = 8;
   p(1:n,1:n+1) = 0.0;  pp(1:n,1:n+1) = 0.0;
   ab = r_jacobi(n,a,b);          % recursion coefficients
   x(:,1) = RootEstimate(n,a,b); % root estimates
   for i=1:MaxNR
      dp = Pcalc(x,ab,p,pp);
      x = x .- dp;
      if(max(abs(dp)) < xtol)break; end
   end
end
function dp = Pcalc(x,ab,p,pp) % Jacobi polynomial p/p'
   p(:,1)  = 1.0;  p(:,2)  = x(:,1) - ab(1,1);
   pp(:,1) = 0.0;  pp(:,2) = 1.0;
   n = size(x,1);
   for k = 2:n
      p(:,k+1) = (x(:,1)-ab(k,1)).*p(:,k) - ab(k,2)*p(:,k-1);
      pp(:,k+1) = p(:,k) + (x(:,1)-ab(k,1)).*pp(:,k) - ab(k,2)*pp(:,k-1);
   end
   dp = p(:,n+1)./pp(:,n+1);
end
```

slow at looping operations. If the computer hardware has vector processing capabilities and the compiler can generate code for it, some dramatic speedups are possible.

There are two problems with the convergence test used in this code. First, a change tolerance like this cannot detect convergence until it has already been achieved. One extra iteration often occurs. To compensate somewhat, the tolerance chosen in the code is three orders of magnitude greater than the machine epsilon. It could likely be much larger still without affecting the results. The second problem is that it is based on the maximum over all the points. If all points have converged except one, then an additional iteration is performed for all the points. The use of Eq. (2.30) for the points near the boundary helps to prevent these wasted iterations. We believe that with the initial estimates implemented, the accuracy would be similar if the convergence test were eliminated and `MaxNR = 2`. This would be the ultimate solution to the deficiencies of the convergence test. Perhaps this idea could be implemented after additional refinement and testing.

The noteworthy features of the iterative technique described here are:

1. Newton-Raphson with vectorized code
2. Accurate initial root estimates
3. Approximately ¼ calculation effort for $\alpha = \beta$

Limited testing of this method suggests that for the most common cases ($\alpha = \beta$), it requires roughly 10 times less computation than a more rudimentary method. The savings are primarily in a reduced number of iterations and symmetric treatment when $\alpha = \beta$. The speedup associated with vectorization will depend on the hardware and compiler/interpreter capabilities. Vectorization is a simple way to gain efficiency, since it only requires one to write clean streamlined code. The trend toward the use of vector hardware is expected to continue.

Fig. 2.4 shows calculated errors in the roots for normal double precision (8 byte) floating point calculations for n to 250. The cases considered are Lobatto shortcut method ($\alpha = 1$, $\beta = \pm½$), Gauss shortcut method ($\alpha = 0$, $\beta = \pm½$), Radau ($\alpha = 1$, $\beta = 0$), full Lobatto ($\alpha = \beta = 1$) and full Gauss ($\alpha = \beta = 0$). The shortcut methods used half the points as described above. The accuracy found for Radau and full calculations with Gauss and Lobatto is independent of n and less than 10^{-16} in agreement with other studies [Hale and Townsend (2013)]. However, there is a small loss of accuracy for use of the shortcut method. This is believed to be due to the square root conversion. See the discussion of Fig. 2.3 for the relationship of the errors. It is likely the dependence shown in Fig.

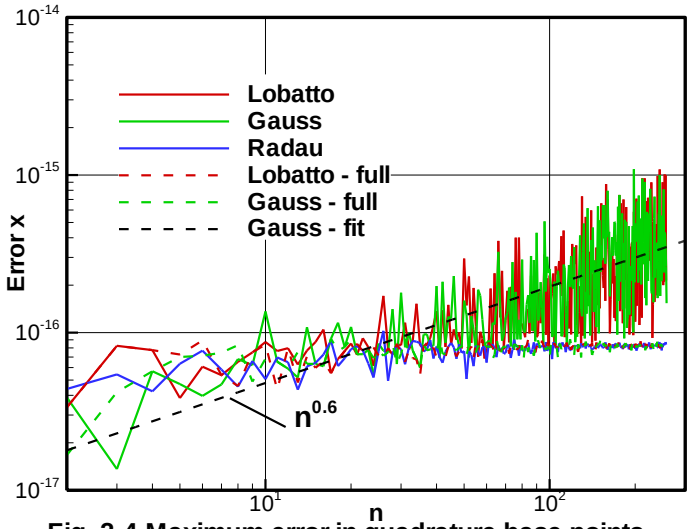

Fig. 2.4 Maximum error in quadrature base points

2.4 is because the smallest root is O(n^{-1}). Even with this slight increase, the errors are about an order of magnitude less than for the popular eigenvalue method, so it is not considered significant for our application.

2.3 Quadrature Weights

The quadrature weights can be calculated directly from the roots of the orthogonal polynomials. Calculation of the weights is discussed in many texts. Frequently one is presented with one formula (and frequently one computer program) for Gaussian quadrature, another for Radau quadrature, a third for Lobatto quadrature and so forth. We will develop all from a single formula. Furthermore, to economize on calculations, all are formulated in terms of the parameters $\hat{p}_n'(x_i)$, i.e. the barycentric weights, which are common to the interpolating polynomials, Eq. (2.2), and other coefficients.

There are many representations for the weights, but the following one is most convenient for our purposes. Krylov (1962, p 113) and Hildebrand (1987, p. 401) give the following relationship for general Gauss-Jacobi weights in Eq. (2.7). The interval [0,1] is used exclusively in this section, so on that basis the weights are:

$$W_i^* = \frac{\Gamma(n+\alpha+1)\Gamma(n+\beta+1)}{\Gamma(n+\alpha+\beta+1)\, n!\, x_i(1-x_i)}\left[P_n^{(\alpha,\beta)\prime}(x_i)\right]^{-2} \tag{2.31}$$

The asterisk denotes the weights for Eq. (2.7), where the integrand contains the weight function $\omega(x) = (1-x)^\alpha x^\beta$. This general formula can be used to calculate the quadrature formulas of interest. Eq. (2.8) shows the Lobatto weights can be calculated by dividing the Gauss-Jacobi weights by $x_i(1-x_i)$. The Radau weights can be calculated in a similar way. For symmetric problems with Lobatto quadrature, the Gauss-Jacobi weight is divided by $(1-x_i^2)$. Using Eqs. (2.2), (2.3) and (2.16), the parameters in Eq. (2.31) are reformulated to use parameters that are common to the interpolation formula and other calculations, so given n, α, β and the roots of the Jacobi polynomial the weights are easily calculated.

Gaussian quadrature uses the roots of Legendre polynomials, α = β = 0, so Eq. (2.31) reduces to:

$$W_i = \frac{[P_n{}'(x_i)]^{-2}}{x_i(1-x_i)} = x_i(1-x_i)[\tilde{\rho}_n \hat{p}_n'(x_i)]^{-2} \tag{2.32}$$

where we note the boundary weights, W_0 = W_{n+1} = 0.

Lobatto quadrature includes weights at both endpoints, so α = β = 1. Following Eq. (2.8) the Lobatto weights are obtained by dividing the Gauss-Jacobi weights by $x_i(1-x_i)$, giving the following:

$$W_i = \frac{(n+1)}{(n+2)}[\tilde{\rho}_n \hat{p}_n'(x_i)]^{-2} \tag{2.33}$$

where we note the boundary weights, W_0 = W_{n+1} > 0.

Radau quadrature includes a weight at only one endpoint. We call it Radau-right if the weight is at x = 1, which calls for α = 1 and β = 0. The Gauss-Jacobi weights, Eq. (2.31), are divided by $(1 - x_i)$ to give the Radau weights:

$$W_i = x_i[\tilde{\rho}_n \hat{p}'_n(x_i)]^{-2} \tag{2.34}$$

For the Radau-left quadrature, the points are the mirror image of the Radau-right roots. The roots with α = 0 and β = 1 are used and the Gauss-Jacobi weights are divided by x_i. The equation for the weights is identical to Eq. (2.34), but with a multiplier of $(1 - x_i)$ appears instead of x_i. The Radau points are easy to calculate, but are of limited usefulness for nonsymmetric problems.

The Gauss-Jacobi quadrature formula can also be used to calculate the weights for symmetric problems, Eq. (2.5). Above we discussed the relationship between symmetric points on [0,1] to the normal ones on [-1,1] and showed how the relationship can be used (when $\alpha = \beta$) to calculate the roots and weights with less effort. To calculate Gaussian quadrature weights for the symmetric case, the roots are calculated with α = 0 and β = -½, 0, ½ for planar, cylindrical and spherical coordinates. Since there is only one endpoint in the symmetric case, the $\hat{p}'_n(x_i^2)$ includes the term $(1 - x_i^2)$, so for Gauss points, the weights are:

$$W_i = \frac{(1 - x_i^2)}{2}[\tilde{\rho}_n x_i \hat{p}'_n(x_i^2)]^{-2} \tag{2.35}$$

The factor of 2 in the denominator is from the conversion to variable x^2 in Eq. (2.5). For Lobatto (actually Radau) weights the Gauss-Jacobi weights are divided by $(1 - x_i^2)$ and the roots are calculated with α = 1 and β = -½, 0, ½. The quadrature weights are:

$$W_i = \frac{n + 1}{2(n + \beta + 1)}[\tilde{\rho}_n x_i \hat{p}'_n(x_i^2)]^{-2} \tag{2.36}$$

A somewhat simpler alternative to the formulas above is to ignore $\tilde{\rho}_n$ and the other constant terms in Eqs. (2.31-35) and determine the proportionality constant from $\sum W_i = 1/(\gamma + 1)$.

The quadrature associated with the Chebyshev points is due to Clenshaw and Curtis (1960). As discussed above and demonstrated in Section 2.9, these roots are not optimal, so the quadrature is less accurate than the choices in Table 2.1. It is an interpolatory quadrature, so Eqs. (2.4) and (2.5) apply. For Cartesian coordinates, formulas for the weights are readily available [Trefethen (2000)]. They can be calculated by either direct integration or by using a fast Fourier transform.

For symmetric planar geometry, it is obvious to use the roots of the even polynomials on (-1,1), i.e. the right half of those used for nonsymmetric problems. Since the use of Chebyshev polynomials is not tied to the accuracy of integration, it is not obvious how to use them for symmetric problems in cylindrical and spherical geometry. We simply use the same points as for planar geometry, so the points are not shifted toward the boundary as they are for Gauss and Lobatto points, see Figs. 1.1, 1.2, 2.6 and 2.7. Calculation of quadrature weights for cylindrical and spherical geometry is also problematic. We calculate them by the direct

integration of Eq. (2.5) using Lobatto quadrature. This is not elegant, but for n interior Chebyshev points, only $(n+1)/2$ interior Lobatto base points are required for the integration.

Fig. 2.5 shows the error in calculated quadrature weights for normal double precision (8 byte) floating point calculations for n to 250. These are the quadrature weights which correspond to the roots shown in Fig. 2.4. The error plotted is the maximum fractional error or what Hale and Townsend (2013) call the maximum relative error. The accuracy found here is comparable to other similar methods and more accurate than methods based on eigenvalue calculations. The smallest weights are O(n^{-2}), so this behavior is not surprising. On the average the weights are O(n^{-1}), so when normalized by the largest weight, the error increases at a slower rate. Fig. 2.5 does not show much difference between the weights calculated with roots from full calculations and those from the shortcut procedure which solves for half the roots (see Eq. (2.22) and associated discussion)

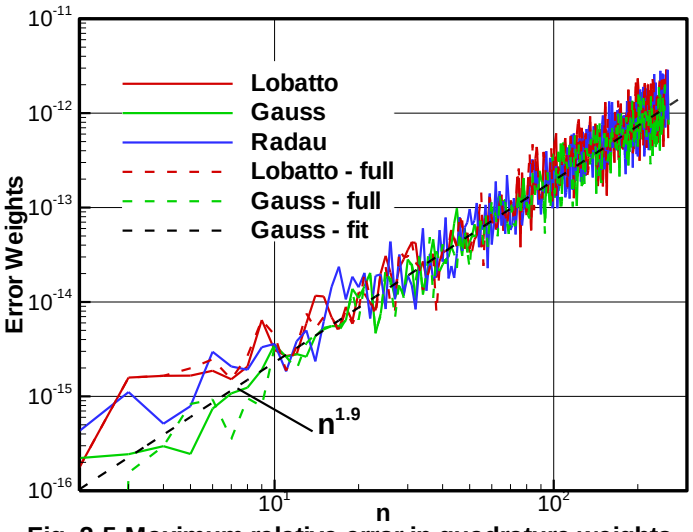

Fig. 2.5 Maximum relative error in quadrature weights

We note that when performing these calculations for very large n the values are combinations of exceedingly small and exceedingly large ones. Consider Eq. (2.32) for Gaussian quadrature. For Legendre polynomials $\tilde{\rho}_{250} = \frac{[2(250)]!}{(250!)^2} \approx 1x10^{150}$, and the $\hat{p}_n'(x_i)$ are correspondingly small so the product produces weights which are O(n^{-1}) on average. The relationships for all weights, Eqs. (2.31-35), involve numbers of similar magnitude. Rather than calculate the constants, our implementation squares the barycentric weights, then uses $\sum W_i = 1/(\gamma + 1)$ to scale the values (see Matlab code for Eq. (2.24)). Normal double precision floating point calculations cannot deal with numbers much greater than $1x10^{300}$, so the limit with our implementation is about $n < 250$. This value is greater than our stated goal. The barycentric weights are also used elsewhere. Eqs. (2.2) and (2.46). Scaling would be relatively easy for the quadrature weights due to the way they are normalized. Scaling would also be easy for the differentiation matrix, Eq. (2.46), since any scaling factors would cancel. Scaling could be more of a problem for the interpolation formula, Eq. (2.2). Hale and Townsend (2013) discuss issues related to calculations with large values of n. The current implementation meets our needs.

2.4 Differentiation of Orthogonal Polynomials

Although a *nodal* formulation is more common, some practitioners use orthogonal polynomial trial functions, Eq. (2.1), where the coefficients are analogous to the modes in a Fourier series.

Here we describe some orthogonal polynomial relationships that are useful for a *modal* formulation with Legendre polynomial basis functions. Other Jacobi polynomials generally have similar or analogous relationships, so a Legendre polynomial basis is considered representative. Chebyshev basis functions are heavily covered elsewhere [Canuto, et al. (1988), Boyd (2000), Shen and Tang (2006)]. All of the discussion here is for the interval [-1,1].

One interesting and useful property of Jacobi polynomials is:

$$\frac{dP_n^{(\alpha,\beta)}(x)}{dx} = \tfrac{1}{2}(n+\alpha+\beta+1)\,P_{n-1}^{(\alpha+1,\beta+1)}(x) \tag{2.37}$$

so the roots of the polynomial on the right corresponds to the extrema of the one indicated on the left. This relationship is apparent in Fig. 2.1. It explains why the Lobatto quadrature base points are often called the extrema of the Legendre polynomials. They are also the roots of the Jacobi polynomials with $\alpha = \beta = 1$. By the same token, the extrema of the Chebyshev polynomials of the 1st kind ($\alpha = \beta = -½$) are the roots of the Chebyshev polynomials of the 2nd kind ($\alpha = \beta = +½$).

In addition to the recurrence relationship, Eq. (2.19), a key identity for Legendre polynomials is:

$$\frac{(1-x^2)}{k}P_k'(x) = P_{k-1}(x) - xP_k(x) \tag{2.38}$$

which can be combined with the recurrence relationship to give:

$$(1-x^2)P_k'(x) = \frac{k(k+1)}{2k+1}\left(P_{k-1}(x) - P_{k+1}(x)\right) \tag{2.39}$$

Eqs. (2.37) and (2.39) can be used to produce the relationship:

$$(1-x^2)P_k^{(1,1)}(x) = -\frac{2k+2}{2k+3}\left(P_{k+2}(x) - P_k(x)\right) \tag{2.40}$$

To use a modal expansion in Legendre polynomials to solve problems with MWR, expressions for the derivatives of Eq. (2.1) are required. From the recurrence relationship for Legendre polynomials Eq. (2.19) and Eq. (2.38) the following can be derived:

$$(2k+1)P_k = P_{k+1}' - P_{k-1}' \tag{2.41}$$

This expression can be applied repeatedly to get one that can be used to calculate the first derivative of Eq. (2.1):

$$P_m' = \sum_{\substack{k=0 \\ k+m\ odd}}^{m-1} (2k+1)P_k(x) \tag{2.42}$$

An expression for the second derivative is:

$$P_m'' = \sum_{\substack{k=0 \\ k+m\ even}}^{m-2} \frac{(2k+1)}{2}\left[m(m+1) - k(k+1)\right]P_k(x) \tag{2.43}$$

Due to the alternating odd/even nature of the polynomials, only alternate values appear in the derivative expressions. The notation *k+m odd* or *even* indicates that only these values appear in the summation. For example:

$$P_5' = P_0 + 5P_2 + 9P_4$$

and

$$P_5'' = 42P_1 + 63P_3$$

A relationship analogous to Eq. (2.42) for any Jacobi polynomial can be derived by starting with the generalization of Eq. (2.41). The generalization of Eq. (2.41) can be derived by combining the derivative of the recurrence relationship, Eq. (2.15), with the recurrence relationship for the derivatives which can be determined using Eq. (2.37):

$$P_k^{(\alpha,\beta)} = a_k^{(\alpha,\beta)} P_{k+1}^{(\alpha,\beta)\prime} - b_k^{(\alpha,\beta)} P_k^{(\alpha,\beta)\prime} - c_k^{(\alpha,\beta)} P_{k-1}^{(\alpha,\beta)\prime} \tag{2.44}$$

where the primes denotes the first derivatives and:

$$a_k^{(\alpha,\beta)} = \frac{2(k+\alpha+\beta+1)}{(2k+\alpha+\beta+1)(2k+\alpha+\beta+2)}$$

$$b_k^{(\alpha,\beta)} = \frac{2(\beta-\alpha)}{(2k+\alpha+\beta)(2k+\alpha+\beta+2)}$$

$$c_k^{(\alpha,\beta)} = \frac{2(k+\alpha)(k+\beta)}{(k+\alpha+\beta)(2k+\alpha+\beta)(2k+\alpha+\beta+1)}$$

Eq. (2.44) can be solved for $P_{k+1}^{(\alpha,\beta)\prime}$ and the right-hand-side terms substituted recursively to calculated the coefficients of:

$$P_n^{(\alpha,\beta)\prime} = \sum_{k=0}^{n-1} d_{nk} P_k^{(\alpha,\beta)} \tag{2.45}$$

Which is analogous to Eq. (2.42), but valid for any Jacobi polynomial.

2.5 Nodal Differentiation Matrices

To develop nodal approximations, derivatives of the basis functions in Eq. (2.2) are required. The first derivative of the Lagrange interpolating polynomials is straight forward. The only trick involved is to take the derivative of the logarithm of Eq. (2.2), thereby converting the continued product to a summation. For the nonsymmetric case, this approach leads to:

$$A_{ij} = \left.\frac{d\ell_j(x)}{dx}\right|_{x_i} = \frac{\hat{p}_n'(x_i)}{(x_i - x_j)\hat{p}_n'(x_j)} \quad \text{for } i \neq j, \text{ and}$$

$$= \sum_{\substack{j=0 \\ j\neq i}}^{n+1} \frac{1}{x_i - x_j} \qquad \text{for } i = j \tag{2.46}$$

and for the symmetric case:

$$A_{ij} = \frac{2x_i \hat{p}_n'(x_i^2)}{(x_i^2 - x_j^2)\hat{p}_n'(x_j^2)} \quad \text{for } i \neq j, \text{ and}$$
$$= \sum_{\substack{j=1 \\ j\neq i}}^{n+1} \frac{1}{x_i^2 - x_j^2} \qquad \text{for } i = j \tag{2.47}$$

These expressions contain $\hat{p}_n'(x_i)$ which are common to the relationships for the interpolating polynomials, $\boldsymbol{\ell}$, and quadrature weights, $\boldsymbol{W}$, so they have multiple uses. As an alternative to the equation above, the diagonal elements can be calculated so the row sums are zero. These expressions are general for any set of points, but simplifications can be made in specific cases. Many texts develop one formula for Gauss points, another for Lobatto points, a third for Chebyshev points and so forth. These distinctions are not necessary, since the formula above is valid for any choice of points, even equally spaced points.

I often see references for Eq. (2.46) that indicate its first use was in the 1980's or later. I first found it in Ferguson (1971) and programmed it in about 1972. It is described by Michelsen and Villadsen (1972). Fornberg (1996) gives the earliest reference I have found, which is Nielsen (1956) (pp. 150-4). It is likely this formula is quite ancient.

For the nonsymmetric case, the second derivative can easily be calculated from the first derivative approximation by a matrix multiply, i.e. $\boldsymbol{B} = \boldsymbol{AA}$, or:

$$B_{ij} = \sum_{k=0}^{n+1} A_{ik} A_{kj} \tag{2.48}$$

For the symmetric cases, the Laplacian operator is not as simple as Eq. (2.48). Since the interpolating polynomial is even, its first derivative is odd. The first derivative operator from Eq. (2.47) is only valid if it operates on a symmetric quantity. Instead, a similar operator is needed for odd functions. The Lagrange interpolating polynomial for an odd function is related to the even function interpolating polynomial by:

$$\hat{\ell}_i = \frac{x}{x_i} \prod_{\substack{j=1 \\ j\neq i}}^{n+1} \frac{x^2 - x_j^2}{x_i^2 - x_j^2} = \frac{x}{x_i} \ell_i(x^2) \tag{2.49}$$

The odd function interpolating polynomial and derivative operator are denoted with a hat (^). The direct differentiation of Eq. (2.49) gives:

$$\hat{A}_{ij} = \frac{1}{x^\gamma} \frac{d}{dx} \left[x^\gamma \hat{\ell}_j \right]_{x_i} = \frac{\gamma + 1}{x_i} \delta_{ij} + \frac{x_i}{x_j} A_{ij} \tag{2.50}$$

The Laplacian operator can now be calculated by $\boldsymbol{B} = \hat{\boldsymbol{A}}\boldsymbol{A}$.

The Laplacian operator, $\boldsymbol{B}$, can also be determined by direct differentiation of Eq. (2.2). For the nonsymmetric case, the values from direct differentiation are:

$$B_{ij} = \frac{2\hat{p}_n'(x_i)}{(x_i - x_j)\hat{p}_n'(x_j)}\left[\sum_{\substack{k=0\\k\neq i\\k\neq j}}^{n+1}\frac{1}{x_i - x_k}\right] = 2A_{ij}\left(A_{ii} - \frac{1}{x_i - x_j}\right) \text{ for } i \neq j, \text{ and}$$

$$B_{ii} = \sum_{\substack{j=0\\j\neq i}}^{n+1}\left[\frac{1}{x_i - x_j}\sum_{\substack{k=0\\k\neq i\\k\neq j}}^{n+1}\frac{1}{x_i - x_k}\right] = A_{ii}^2 - \sum_{\substack{j=0\\j\neq i}}^{n+1}\frac{1}{x_i - x_j} \tag{2.51}$$

Similar relationships can be developed for the symmetric case.

In finite element methods an alternate approximation is used for the Laplacian. It is called the *stiffness matrix*, which reveals the roots of these methods in structural mechanics. However, it is a general alternative representation of the Laplacian in *weak* form. We generalize it to some extent for use with collocation. It is easily calculated from the other arrays. For nonsymmetric problems:

$$\begin{aligned} C_{ij} &= \delta_{i,n+1}A_{n+1,j} - \delta_{i,0}A_{0,j} - W_iB_{ij} \\ &= \sum_{k=0}^{n+1} W_kA_{ki}A_{kj} = \int_0^1 \frac{d\ell_i(x)}{dx}\frac{d\ell_j(x)}{dx}dx \end{aligned} \tag{2.52}$$

For symmetric problems there is no boundary point at $i = 0$, so those terms are omitted and the $A_{0,j}$ term does not appear in Eq. (2.52). As explained in the examples, the top expression amounts to a simple rearrangement of the basic equations, so it is valid for any set of points. The validity of the second set of equalities depends on the accuracy of the quadrature. It is always valid for Lobatto and Radau points and is also valid for Gauss points in symmetric problems. For other cases, the top expression must be used to calculate the stiffness matrix. Nevertheless, the stiffness matrix is symmetric for Gauss, Radau and Lobatto points and for small n with Chebyshev points.

The *mass matrix* is another useful matrix with a name from structural mechanics. Of course, for many applications it has nothing to do with mass. Its use is explained in the examples. For the Moments method it is:

$$M_{ij} = \int_0^1 \ell_i^*(x)\ell_j(x)f(x)dx \tag{2.53}$$

where $\ell_i^*(x) = \ell_i(x)x_i(1 - x_i)/[x(1 - x)]$ are reduced interpolating polynomials which go through the interior points only. The corresponding matrix for a Galerkin method uses the full interpolating polynomial, $\ell_i(x)$. $f(x)$ is a coefficient of the problem. For problems with symmetry and $f(x) = 1$, the associated quadrature (Gaussian for Moments, Lobatto for Galerkin) is exact, so Eq. (2.53) reduces to:

$$M_{ij} = \delta_{ij}W_if(x_i) \tag{2.54}$$

For nonsymmetric problems $\boldsymbol{M}$ is a full matrix. If $f(x) = 1$, it is calculated using a quadrature formula with one additional base point. For a more general function, it can be approximated by a number of extra quadrature points which may be specified. Note that $\boldsymbol{M}$ is symmetric for the Galerkin method and nonsymmetric for Moments.

2.6 Discrete Jacobi Transforms

As stated in Chapter 1, there is a simple relationship between the nodal, modal, and monomial representations. Modal and nodal approximations are given by Eqs. (2.1) and (2.2), respectively, while a monomial representation is given in Eq. (2.66) in the next section. Apart from possible differences in machine roundoff, there is no difference in the computed solution when any of the three representations is used for the trial solution. Although this monograph uses a nodal representation, transforms with the other representations are described for completeness. Here we examine the relationship between nodal and modal representations, while below in Section 2.7 transforms between nodal and monomial representations is described.

If one is given the modal coefficients, $\hat{\boldsymbol{a}}$, in Eq. (2.1), it is straight forward to compute the nodal values by simply summing the product of the coefficients and the polynomial values at the nodes. However, suppose the nodal values, $\boldsymbol{y}$, are known and we wish to determine the modal coefficients which interpolate $y(x)$. This problem is not as simple. One approach would be to treat the problem as a linear algebraic system to solve for the modal coefficients. The algebraic approach is particularly problematic with monomials, since the resulting Vandermonde matrix is notoriously ill conditioned. However, there is an easier way.

The coefficients $\hat{\boldsymbol{a}}$ can be calculated using a discrete Legendre transform, which is analogous to a Fourier transform, so the coefficients can be calculated with a similar procedure:

$$\hat{a}_\ell = \frac{1}{\zeta_\ell}\sum_{i=0}^{n+1} y(x_i)\int_{-1}^{1} \ell_i(x)P_\ell(x)dx = \sum_{i=0}^{n+1} Q_{\ell i}y(x_i) \tag{2.55}$$

where $\zeta_\ell = 2/(2\ell+1)$ as indicated in Eq. (2.17) and $\boldsymbol{Q}$ is the transformation matrix. If the integrals are evaluated with the quadrature that is associated with the points, then the transformation matrix is given by:

$$Q_{\ell i} = \frac{1}{\hat{\zeta}_\ell}\sum_{k=0}^{n+1} W_k \ell_i(x_k)P_\ell(x_k) = \frac{W_i}{\hat{\zeta}_\ell}P_\ell(x_i) \tag{2.56}$$

There are n interior nodes and two boundary nodes in Eq. (2.55), so the integrand consists of two polynomials up to n + 1 degree for a total of up to $2n$ + 2 degrees. Lobatto quadrature is exact for $2n$ + 1, which misses exact integration by one degree, so all but $\hat{a}_{n+1}$ will be calculated correctly. Gaussian quadrature is exact for $2n$ - 1, so the last three $\hat{a}$ will be in error. However, Shen and Tang (2006) show that Lobatto quadrature can be made to work by using the following modification:

$$\hat{\zeta}_\ell = \begin{cases} \zeta_\ell & \text{for } \ell < n+1 \\ \zeta_\ell \left(2 + \dfrac{1}{\ell}\right) & \text{for } \ell = n+1 \end{cases} \tag{2.57}$$

Gaussian quadrature cannot be made to work with this approach unless additional quadrature points are used in the integration of Eq. (2.55). Gauss points require the far more cumbersome calculation of the integrand at more than n base points, so the simplification to Eq. (2.56) does not occur.

Many texts [e.g. Canuto, et al. (1988), Boyd (2000)] discuss the use of Eq. (2.56), but they fail to consider the endpoints, which are required to meet the boundary conditions, even for Gauss points.

If the problem has homogenous Dirichlet boundary conditions it is possible to expand the solution in Jacobi polynomials which can then be converted to a Legendre series using Eq. (2.40) as follows:

$$y(x) = (1-x^2)\sum_{k=0}^{n-1} b_k P_k^{(1,1)}(x) = -\sum_{k=0}^{n-1} b_k \left(\frac{2k+2}{2k+3}\right)\left(P_{k+2}(x) - P_k(x)\right) \tag{2.58}$$

where the coefficients are:

$$b_\ell = \frac{1}{\zeta_\ell^{(1,1)}} \sum_{i=1}^{n} y(x_i) \int_{-1}^{1} \ell_i(x) P_\ell^{(1,1)}(x)\,dx = \sum_{i=1}^{n} Q_{\ell i}^J y(x_i) \tag{2.59}$$

and from Eq.(2.6), $\zeta_\ell^{(1,1)} = 8(\ell+1)/[(2\ell+3)(\ell+2)]$. The Jacobi transform for $\alpha = \beta = 1$ requires the integrand to contain the weight function, $(1 - x^2)$. Eq. (2.59) appears to violate this requirement, but in fact the term is contained within the interpolating polynomial.

If we use the quadrature associated with the points selected, $\boldsymbol{Q}^J$ is given by:

$$Q_{\ell i}^J = \frac{W_i}{\hat{\zeta}_\ell^{(1,1)}} P_\ell^{(1,1)}(x_i) \tag{2.60}$$

$\ell = 0,\ldots,n-1$ and $i = 1,\ldots,n$. With this approach, the integrand in Eq. (2.59) is two degrees less than the one in Eq. (2.55), a total of up to $2n$ degrees. Lobatto quadrature is exact, while Gaussian quadrature misses exact integration by one degree. However, an exact result is obtained by using the following for Gauss points:

$$\hat{\zeta}_\ell^{(1,1)} = \begin{cases} \zeta_\ell^{(1,1)} & \text{for } \ell < n-1 \\ \zeta_\ell^{(1,1)} \left(\dfrac{2\ell+3}{\ell+2}\right) & \text{for } \ell = n-1 \end{cases} \tag{2.61}$$

The modification above is not needed for Lobatto points, since they produce an exact result.

The development using Eqs. (2.58) and (2.59) assumes homogenous boundary values, $y_0 = y_{n+1} = 0$; however, it is easy to generalize to arbitrary endpoint values by using an expansion of the form:

$$y(x) = \frac{(1-x)}{2}y_0 + \frac{(1+x)}{2}y_{n+1} + (1-x^2)\sum_{k=0}^{n-1} b_k P_k^{(1,1)}(x)$$
$$= (y_0 + y_{n+1})\frac{P_0}{2} - (y_0 - y_{n+1})\frac{P_1}{2} + \sum_{k=0}^{n-1} a_k (P_{k+2}(x) - P_k(x)) \quad (2.62)$$

The values of the coefficients, $\boldsymbol{b}$, are determined using Eq. (2.59), with $y(x_i) - 0.5(1-x_i)y_0 - 0.5(1+x_i)y_{n+1}$ substituted for $y(x_i)$. Next, define an intermediate transform matrix for the coefficients $\boldsymbol{a}$:

$$\tilde{Q}_{k,0} = \frac{1}{2}\sum_{i=1}^{n}\left(\frac{2k+2}{2k+3}\right)Q_{ki}^{J}(1-x_i)$$
$$\tilde{Q}_{kj} = -\left(\frac{2k+2}{2k+3}\right)Q_{kj}^{J} \quad (2.63)$$
$$\tilde{Q}_{k,n+1} = \frac{1}{2}\sum_{i=1}^{n}\left(\frac{2k+2}{2k+3}\right)Q_{ki}^{J}(1+x_i)$$

for k = 0,…,n - 1 with $\tilde{Q}_{kj} = 0$ for $k = -2, -1, n, n+1$. The intermediate transform in Eq. (2.63) can be used to calculate the coefficients of Eq. (2.62) by summing over the boundary and interior points:

$$a_k = \sum_{i=0}^{n+1} \tilde{Q}_{ki} y(x_i) \quad (2.64)$$

The Legendre transformation matrix in Eq. (2.55) can then be completed by collecting like terms:

$$Q_{kj} = \tilde{Q}_{k-2,j} - \tilde{Q}_{kj} + \tfrac{1}{2}(\delta_{k,0} - \delta_{k,1})\delta_{j,0} + \tfrac{1}{2}(\delta_{k,0} + \delta_{k,1})\delta_{j,n+1} \quad (2.65)$$

for k = 0,…,n + 1. The transform matrix defined by Eq. (2.65) is identical to that defined by the Legendre transform in Eqs. (2.55) and (2.56).

In summary, the Legendre transform matrix, $\boldsymbol{Q}$, can be calculated quite simply for either Gauss or Lobatto interior points and the boundary points. A simple matrix multiply, Eq. (2.55), gives the coefficients of the truncated Legendre series, Eq. (2.1), which interpolates the nodal values. Transforming in the other direction is even easier. Given the modal coefficients, the nodal values are calculated by summing the coefficients times the polynomials evaluated at the nodes, i.e. evaluation of Eq. (2.1) at the nodes.

As explained by Boyd (2000), the transform matrix, $\boldsymbol{Q}$, can be substituted into any of the approximations to convert from one form to the other, i.e. modal to nodal or vice versa. For example, suppose a matrix $\boldsymbol{B}$ acts on modal coefficients. It can be converted to one for nodal values by: $\boldsymbol{B}\hat{\boldsymbol{a}} = \boldsymbol{BQy}$. As an example we convert the modal differentiation matrix given by Eq. (2.42) to a nodal matrix as follows:

$$\sum_{k=0}^{n+1} \left.\frac{dP_k(x)}{dx}\right|_{x_i} Q_{kj} = A_{ij} = \left.\frac{d\ell_j(x)}{dx}\right|_{x_i}$$

where $\boldsymbol{A}$ is the first derivative matrix given by Eq. (2.46). As an example, consider Gauss points with n = 2. Eq. (2.42) is evaluated at the endpoints ±1 and the two interior points ±0.57735 for the first four polynomials. Post multiplication by the transformation matrix gives:

$$\begin{bmatrix} 0 & 1 & -3.00000 & 6 \\ 0 & 1 & -1.73205 & 1 \\ 0 & 1 & 1.73205 & 1 \\ 0 & 1 & 3.00000 & 6 \end{bmatrix} \begin{bmatrix} 0.00 & 0.50000 & 0.50000 & 0.00 \\ -0.20 & -0.51962 & 0.51962 & 0.20 \\ 0.50 & -0.50000 & -0.50000 & 0.50 \\ -0.30 & 0.51962 & -0.51962 & 0.30 \end{bmatrix} = \begin{bmatrix} -3.50000 & 4.09808 & -1.09808 & 0.50000 \\ -1.36603 & 0.86603 & 0.86603 & -0.36603 \\ 0.36603 & -0.86603 & -0.86603 & 1.36603 \\ -0.50000 & 1.09808 & -4.09808 & 3.50000 \end{bmatrix}$$

After multiplying by 2 to rescale from [-1,1] to [0,1], the values for $\boldsymbol{A}$ are identical to those from Eq. (2.46) which is implemented in supplied code. This example is also listed in Finlayson (1972), Table 5.5. Some additional conversions of this type are illustrated in the examples.

Some algorithms [Canuto, et al. (1988) p. 86 and Boyd (2000), p. 107] require switching back and forth between the modal and nodal representations while solving some nonlinear problems. In such cases, the efficiency of the transformation can become important. In most cases the transformation matrix has special form which can be exploited to speed up the calculation. For Chebyshev trial functions, the transformation can be performed using fast Fourier transforms, which is the most efficient method for very large n, i.e. n > 20 – 100. Our preference is to use a nodal formulation which is much simpler for nonlinear problems and if large n is required then switch to a nodal finite element based method. If a problem truly benefits from use of a high order method, one could always use say 6 interior nodes per element to achieve an 8th order convergence rate!

Many texts suggest or at least imply the form of the trial functions has a major effect on the results, the example calculations illustrate that, with regard to the calculated solution, it makes absolutely no difference whether the approximation is nodal, modal or monomials. The results are always equivalent and can easily be converted from one form to another. Rounding errors are the only potential source of differences.

2.7 Monomial Transforms

Early deveopments of Orthogonal Collocation did not calculate the nodal differentiation matrices as described in Section 2.5. In the early descriptions [Villadsen and Stewart (1967), Finlayson (1972)], orthogonal polynomials were stated to be the trial functions. Despite these statements to the contrary, the approximations were developed using monomial trial functions:

$$y(x) \approx \sum_{k=0}^{n+1} \breve{a}_k x^k \tag{2.66}$$

The monomials were differentiated to develop expressions for derivatives and a transformation matrix converted the results to nodal approximations. For example, the first derivative matrix was calculated with an expression like:

$$\left.\frac{dy}{dx}\right|_{x_i} \approx \sum_{k=0}^{n+1} k\, x_i^{k-1}\, \breve{a}_k = \sum_{j=0}^{n+1}\left(\sum_{k=0}^{n+1} k\, x_i^{k-1}\breve{Q}_{jk}\right) y(x_j) = \sum_{j=0}^{n} A_{ij}\, y(x_j) \tag{2.67}$$

The other differentiation matrices and the quadrature points were calculated with a similar procedure, i.e. by differentiating or integrating the monomials followed by transformation. The transformation matrix, $\breve{\boldsymbol{Q}}$, was calculated by inverting the Vandermonde matrix, which is notoriously ill conditioned, so the approach is only valid for $n \lesssim 10$. We will develop a more direct procedure for calculating the transformation matrix.

The previous section discussed the transformation between nodal and modal representations of the solution. Here we consider the analogous transformation between nodal and monomial representations. Although this monograph uses nodal approximations, the transforms to other formulations are described for completeness. If the coefficients of Eq. (2.66) are known, it is easy to determine the nodal values by direct substitution. It is more difficult to determine the coefficients from the nodal values. We need a transformation analogous to Eq. (2.55):

$$\breve{a}_\ell = \sum_{k=0}^{n+1} \breve{Q}_{k\ell} y(x_k) \tag{2.68}$$

The coefficients are those that interpolate through the nodal values, so what is needed is an expansion of the interpolating polynomial in Eqs. (2.2). Combining Eqs. (2.2) and (2.68) gives the following relationship:

$$\ell_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{n+1} \frac{(x - x_j)}{(x_i - x_j)} = \sum_{k=0}^{n+1} \breve{Q}_{ik} x^k \tag{2.69}$$

The values of $\breve{\boldsymbol{Q}}$ can be calculated by expanding the continued product. The denominator is a simple calculation. The expansion of a continued product can be accomplished as follows:

$$\prod_{i=0}^{n+1}(x - x_i) = (x - x_0)\prod_{i=1}^{n+1}(x - x_i) = \left(x^k + \sum_{j=0}^{k-1} b_{jk} x^j\right)\prod_{i=k}^{n+1}(x - x_i) \tag{2.70}$$

The polynomial in the large parenthesis is built up recursively. The second equality shows values for k = 1. When k is incremented, the coefficients are updated by $b_{i,k+1} = b_{i-1,k} + (-x_k)b_{ik}$ with $b_{kk} = 1$.

Perhaps it is easier to understand the procedure by examining some Matlab code. Referring to the text box, `x` are the points and `n` is the total number of points. The first set of statements creates an (`n`)x(`n-1`) array, `xj`, which excludes the point corresponding to the row number and calculates the continued product for the denominator of Eq. (2.69). The second set of nested loops recursively calculates the values for each coefficient by expanding the continued product in the numerator.

We demonstrate the transform by using Eq. (2.67) to calculate the same first derivative matrix considered in the previous section, i.e. for 2 interior Gauss points. The first matrix holds the

Monomial Transform Calculation

```
xj = repmat(x(2:end)',n);
for i=1:n
   xj(i,1:i-1) = x(1:i-1)';
   q(i,1) = ((1.0)./prod(x(i)-xj(i,:)));
end
for k=2:n
   q(:,k) = q(:,k-1);
   for kk=k-1:-1:2
      q(:,kk) = q(:,kk-1) .- xj(:,k-1).*q(:,kk);
   end
   q(:,1) = -xj(:,k-1).*q(:,1);
end
```

values of the derivatives $j\,x_i^{j-1}$ at the two endpoints and two interior points 0.2113 and 0.7887. The second matrix is the transpose of $\breve{\boldsymbol{Q}}$, and the result on the right-hand-side is the first derivative matrix, $\boldsymbol{A}$. The values agree with those in the furnished software and are twice those in the previous section, since those are on the interval [-1,1] and these are on [0,1].

$$\begin{bmatrix} 0 & 1 & 0.0000 & 0.000 \\ 0 & 1 & 0.4226 & 0.134 \\ 0 & 1 & 1.5774 & 1.866 \\ 0 & 1 & 2.0000 & 3.000 \end{bmatrix} \begin{bmatrix} 10 & 0.000 & 0.000 & 0 \\ -7 & 8.196 & -2.196 & 1 \\ 12 & -18.588 & 12.588 & -6 \\ -6 & 10.392 & -10.392 & 6 \end{bmatrix} = \begin{bmatrix} -7.000 & 8.196 & -2.196 & 1.000 \\ -2.732 & 1.732 & 1.732 & -0.732 \\ 0.732 & -1.732 & -1.732 & 2.732 \\ -1.000 & 2.196 & -8.196 & 7.000 \end{bmatrix}$$

2.8 Software

Stroud and Secrest (1966) have tabulated the roots and quadrature weights to high accuracy for numerous cases through large n. Rather than using tabular values from a book, it is more convenient to have a computer code which will produce the desired quantities when requested. The computer code could store the results internally, calculate the roots and all of the coefficients, or some combination of the two approaches.

Normally, the coefficient calculations described above make up a small portion of the total calculations required to solve a problem. However, there is much debate in the literature regarding the efficiency of the calculation of the points, quadrature weights, interpolating polynomials and various differentiation matrix quantities described here. Central to the calculation of these quantities are the calculation of the points or roots of the specific polynomials. This issue is trivial for the Chebyshev points since they can be calculated directly using Eq. (2.21). The associated Clenshaw-Curtis quadrature weights can also be calculated efficiently. For the other polynomials specified in Table 2.1, there are basically three options:

1. Store precalculated roots and optionally other quantities
2. Calculate roots iteratively and weights and other quantities as described above
3. Calculate roots and weights by solving the tridiagonal eigenvalue problem constructed from the recurrence relation

The first option is listed, because it seems to get lost in the debate. “Back in the day”, we calculated the quantities of interest and punched the results out on cards, which were read by our application codes. A modernized version of this approach is still an option if one is concerned about the cost of these calculations.

There are also variations on these three basic methods, e.g. calculate roots from the eigenvalue problem and quadrature weights from Eqs. (2.31-35). This particular option makes a lot of sense. To solve an application problem, one usually requires at a minimum $\boldsymbol{W}$, $\boldsymbol{A}$ and $\boldsymbol{\ell}$, all of which depend on the so called barycentric weights, $1/\hat{p}_n'(x_i)$. The calculation of $\boldsymbol{W}$ and $\boldsymbol{A}$ is trivial given these quantities, see Eqs. (2.31-35), (2.46) and (2.47).

Both methods 2 and 3 listed above are equivalent. The iterative approach finds the roots of the characteristic equation of the eigenvalue problem. There are many public domain computer codes available using both methods. Hale and Townsend (2013) have compared some of these and found that both methods normally require calculations and computing time of roughly O(n^2). However, they did not compare different implementations of the same basic method, which can sometimes make a significant difference. For example, as discussed in Section 2.2, we achieved roughly a factor of 10 improvement to an iterative code with some rather simple modifications. The method is still O(n^2). The calculation of $\hat{p}_n'(x_i)$ and most of the derivative matrices are also O(n^2). For comparison, a simple matrix-vector multiply requires O(n^2) calculations, while a matrix-matrix multiply or solving an n x n algebraic system by direct elimination requires O(n^3) operations.

Several authors have provided software for calculating some of the fundamental quantities discussed in this chapter. Villadsen and Michelsen (1978) and Fornberg (1996) include FORTRAN 77 source code listings in their books. Source code can be downloaded from several sites.

Funaro (1992) FORTRAN 77 (80 programs, 3500 lines):
morespace.unimore.it/danielefunaro/routines/
Don, Somonoff and Costa Fortran 90 PseudoPack high performance software:
www.cfm.brown.edu/people/wsdon/PseudoPack.htm
Weideman and Reddy (2000) Matlab Differentiation Suite:
appliedmaths.sun.ac.za/~weideman/research/differ.html.
Trefethen (2000) software from book:
people.maths.ox.ac.uk/trefethen/spectral.html
Shen, Tang and Wang (2011) approximately 50 Matlab functions from book:
www.ntu.edu.sg/home/lilian/book.htm
Gautschi’s (2005) OPQ Suite, orthogonal polynomials and quadrature:
cs.purdue.edu/archives/2002/wxg/codes/OPQ.html
Burkardt (2015), quadrature, numerous other codes in Fortran 90, C++ and Matlab:
people.sc.fsu.edu/~jburkardt/
Young (2017) from this ebook in Matlab/Octave, Fortran 90 and C++ :

tildentechnologies.com/Numerics/

Some of this software can do most of the calculations of interest here. Some provide results only for Chebyshev points or cannot do calculations for all the points of interest, e.g. symmetric problems. Many are associated with books and require a knowledgeable user. Most are not written in an object oriented style with a modern interface. Other than the software developed here and the quadrature software described below, I have not found a great deal of code in C++.

There are many codes available for calculating only the polynomial roots and quadrature weights, but no differentiation matrices, etc. Stroud and Secrest (1966) used an iterative method to solve for the polynomial roots. Their code is still available. Elkay and Kautsky (1987) developed a very general FORTRAN 77 code based on the Golub and Welsh method (discussed in Section 2.2). GAUSSQ (1975) is another FORTRAN 77 code based on that method. Burkardt (2015) has adapted many of these old FORTRAN 77 codes to Fortran 90, C, C++ and Matlab. Gautschi's (2005) OPQ Suite does Matlab calculations with orthogonal polynomials, including quadrature formulas. Many of these codes can calculate a number of other quadrature formulas, as a result some are quite large.

We have not attempted to make an exhaustive comparison of these codes. Those tested seem to work well, but they are not written in an object oriented style. We have developed computer codes in Fortran 90/95, C++ and Matlab/Octave. Some have used public domain software for the roots and quadrature weights. The other quantities are calculated using the relationships described above. The Fortran 90 code is encapsulated in a module and the C++ uses a class. A modern interface is then used to access the quantities of interest. The data and internal procedures are protected using the private attribute. The internal calculations could easily be changed without affecting the public interface used by applications. There has been no concerted effort to make this code efficient, although it does store and reuse the $\hat{p}_n'(x_i)$ when possible. I have chosen to emphasize code structure and ease of use. There is likely some room for efficiency improvement.

The current Fortran 90 module, *OCC.f90*, obtains the Jacobi roots using the vectorized iterative method described in Section 2.2. This code uses accurate initial estimates using methods not available at the time much of the legacy software was written. The module calculates the quadrature weights and various matrix operators using array valued functions. It would be relatively easy to substitute a different code for finding the roots without changing the external interface, e.g. the Elkay and Kautsky code for roots and weights. The module also supports Radau points and Chebyshev points (Clenshaw-Curtis quadrature). Chebyshev points in symmetric cylindrical and spherical geometry are supported where the quadrature weights are calculated by direct integration of Eq. (2.5). There is also an experimental Fortran module, *OrthPoly.f90*, for calculating orthogonal polynomial properties for modal methods.

For use with Excel, a dynamic link library was created from an earlier version of the Fortran 90 code using simple nonmodule Fortran routines and Visual Basic to simplify the interface. The dynamic link library lacks some of the features of the current Fortran 90 version.

The C++ code is more basic than the Fortran code. It is based on the *Jacobi_Rule* code from Elkay and Kautsky translated by Burkhardt (2015). A wrapper was used to encapsulate that code within a C++ class and to provide a modern interface. The class also calculates the interpolating polynomials and the various matrix operators calculated using the relationships given above. We plan to replace the Elkay and Kautsky code which is quite large (>1000 lines) by the more compact and efficient vectorized iterative algorithm in Section 2.2. We also plan to add calculations for Chebyshev points and Clenshaw-Curtis quadrature, which are not currently supported.

We do not have access to Matlab so use the free clone, Octave, instead. We originally used the C++ code to create a Mex (Matlab EXecutable) file. Although Mex files are claimed to be compatible, we have discovered they are not, so we have given up on that approach.

We now have native Matlab/Octave .m files. By using Gautschi's (2005) OPQ *r_jacobi* function to calculate the recurrence relations, we created a compact function to calculate $\boldsymbol{x}$, $\boldsymbol{W}$ and $\boldsymbol{A}$ for Gauss, Lobatto and Radau quadrature points. This function can obtain the roots with either of the codes shown in Section 2.2, the eigenvalue code or the vectorized iterative code. Another short function was created for symmetric problems and a third calculates the second derivative matrices, $\boldsymbol{B}$ and $\boldsymbol{C}$. These three programs total about 50 lines of code if the eigenvalue method is to calculate the roots. I would never have believed this was possible (see text boxes in Section 2.2). Combining these routines for Gauss, Lobatto and Radau points with the Matlab software for Chebyshev points (see example below) gives a simple native Matlab code for most cases one might consider. Although this code is a study on compact coding, the iterative method, including the shortcut for $\alpha = \beta$ and accurate initial estimates, is more accurate and efficient than use of the *eig* function. However these refinements require an additional 65 lines of code. These functions have been supplemented by others to calculate full mass matrices, Eq. (2.53), Lagrange interpolating polynomials, Eq. (2.2), and monomial transforms, Eq. (2.69).

Matlab/Octave software for Clenshaw-Curtis quadrature is readily available: an FFT based code from von Winckel (2004) and a conventional code from Trefethen (2000). Trefethen also gives calculation of the derivative operator in Matlab. These have been extended to calculate quantities for symmetric problems. The quadrature weights for symmetric problems in cylindrical and spherical coordinates are calculated using the monomial transforms.

The various codes provide a toolkit for solving problems with not only orthogonal collocation but also other methods of weighted residuals in a wide range of languages and calculation systems. Some of the examples demonstrate use of these codes with Galerkin or Moments methods. A spreadsheet is not an ideal platform for solving differential equations, but we have

found it very useful for checking the calculations for correctness. The example problems usually create tab delimited files for easy spreadsheet importation.

The codes calculate the following quantities:

1. Base points, $\boldsymbol{x}$
2. Quadrature weights, $\boldsymbol{W}$
3. Lagrange interpolating polynomials, $\boldsymbol{\ell}(x)$
4. First derivative operators, $\boldsymbol{A}$ and $\widehat{\boldsymbol{A}}$
5. Laplacian operator, $\boldsymbol{B}$
6. Symmetric Laplacian or stiffness matrix, $\boldsymbol{C}$
7. Mass matrix for Galerkin or Moments methods, $\boldsymbol{M}$
8. Monomial transform, $\boldsymbol{Q}$

I have made every effort to demonstrate good programming style for all the code in this monograph. The technique of encapsulation by using a wrapper in a Fortran module or C++ class is one example. This technique has been used not only for quadrature, but also with public domain linear algebra codes for solving matrix and eigenvalue problems. As a simple example, we use a parameter `float` to allow easy precision changes in Fortran. This made it easy to compare the accuracy of points and weights in Figs. 2.4 and 2.5. I frequently see code with hardwired "`kind = 8`" throughout, making precision changes a daunting and error prone task. There is a huge inventory of tried and true Fortran code in the public domain. The technique of encapsulation to reuse the code with a modern interface and protection is one every coder should learn. It is relatively easy to mix languages using interlanguage calling, common libraries, and conversion utilities like *f2c*, *f2f90* and *f2matlab*. With these tools one can incorporate tried and true public domain code using languages besides Fortran. The dynamic link library for Excel and Mex files for Matlab are two examples. Although the example problems are small, the use of protection for data and internal calculations demonstrates how to make distinct interfaces between program components for more reliable programs. I have found a guide for fortran and mixed language which I mostly agree with at - http://www.fortran90.org/src/best-practices.html.

2.9 Example Calculations

For each of the four language systems implemented (C++, Fortran 90, Octave/Matlab and Excel) a simple test program is provided to demonstrate the call syntax and use of the quantities calculated. Codes for the example problems are also provided. The code and a simple reference manual can be freely downloaded.

This section contains excerpts from the test codes and their results. The examples show some of the basic calculations and demonstrate their use for interpolation, differentiation and integration. For a more detailed understanding, one should experiment with the test codes. The example code below is in Fortran 90 and Matlab/Octave, since either one can produce compact code using array syntax. For C++ loops are required to produce the same results, so

many more statements are needed. The C++ code and test spreadsheet perform these same calculations.

The Fortran code first requires declaration of all the variables and the *Use* statement to include the module which performs the calculations. This module also has some useful names and text strings to improve code readability and usability. It is better to use the *only* variant of the *Use* statement for documentation and to include only the programs and data of interest. In the test code almost everything in the module is used. Next, `CollocSet` is called to specify the symmetry, type (Gauss, Lobatto, etc.) and the geometry. All arguments to `Collocset` are optional, so this call uses keywords which may be in any order. Then one function is called for each of the desired arrays. Another procedure (`CollocCoef`) is provided to calculated $\boldsymbol{x, w, A}$ and optionally $\boldsymbol{B}$ and $\boldsymbol{C}$ with one call. Two versions of the `MassMatrix` function are demonstrated, one assumes no function in Eq. (2.53), while the other gives the name of the function to use. An array of values, `xi` = (0, 0.05, 0.10….), has been set up for tabulation and

Fortran Example Code

```
Use OrthogonalColloc
Call CollocSet(Symmetry=Symmetric,TType=typ,Geometry=igeom)
x = Xpoints(n)
w = Weights(x)
A = Acoeff(x)
if(Symmetric)An = AnonSym(x,A)
B = Bcoeff(x,A)
C = Ccoeff(x,w,A)
D = MassMat(x)
Df = MassMat(x,MassFunc,2)
Li = Lcoeff(xi,x)
Q = Lpoly(x)
call ExpFunc(x,fx,fc,dfc)
call ExpFunc(xi,fx,f,df)
fi = MatMul(Li,fc)
dfa = MatMul(A,fc)
fw = Sum(w*fc)
```

Matlab Example Code

```
[x,w,A,An] = OCsym(n,meth,geom);
[B,C] = OCBCcoef(w,A,An,symm);
D = MassMatrix(x);
Df = MassMatrix(x,@MassFunc,2);
Li = Lcoef(xi',x);
Q = Lpoly(x,symm);
for i=1:nt
   Lp(:,i) = polyval(fliplr(Q(i,:)),xi);
end
[fx,fc,dfc] = Expfunc(x,symm,geom);
[fx,f,df] = Expfunc(xi,symm,geom);
fi = Li*fc;
dfa = A*fc;
fw = w'*fc;
```

plotting values. The `Lcoef` function calculates values of the interpolating polynomials, while `Lpoly` calculates the monomial coefficients of the polynomials. The routine `ExpFunc` calculates the values, derivative and integral of $exp(-5x^2)$ at the collocation points and `xi`. The last three statements interpolate the function and approximate its first derivative at the collocation points, and then calculates an approximate integral using the quadrature formula.

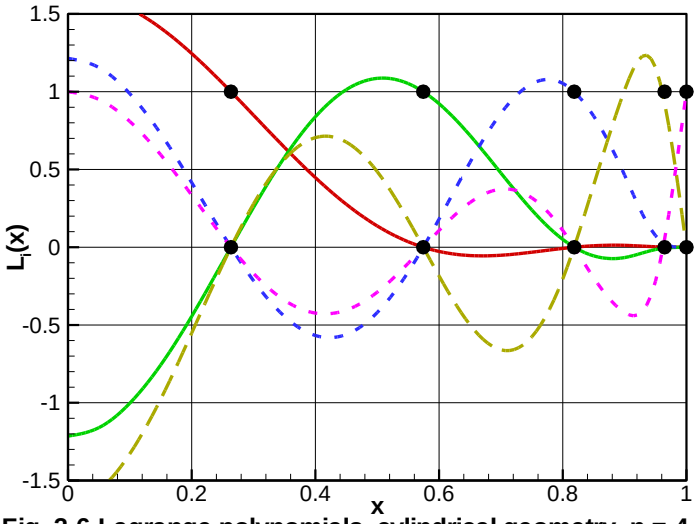

Fig. 2.6 Lagrange polynomials, cylindrical geometry, n = 4

The second text box contains code for Matlab which performs the same tasks. All of these calculations are with native Matlab code. The last three statements perform the same calculations approximating the first derivative, integral and interpolation.

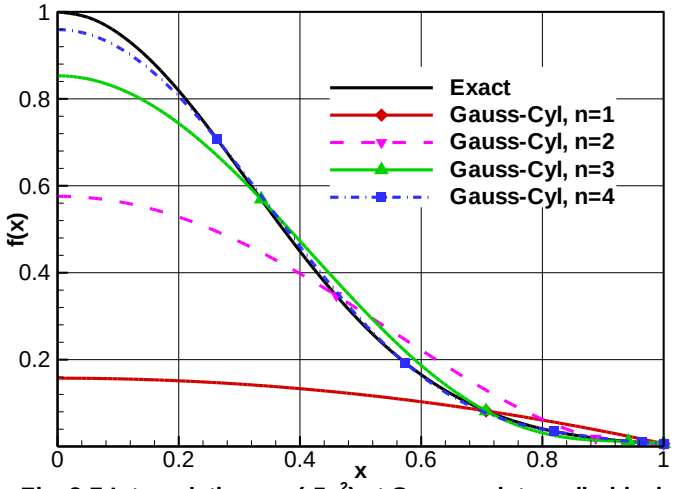

Fig. 2.7 Interpolation exp(-5x^2) at Gauss points, cylindrical

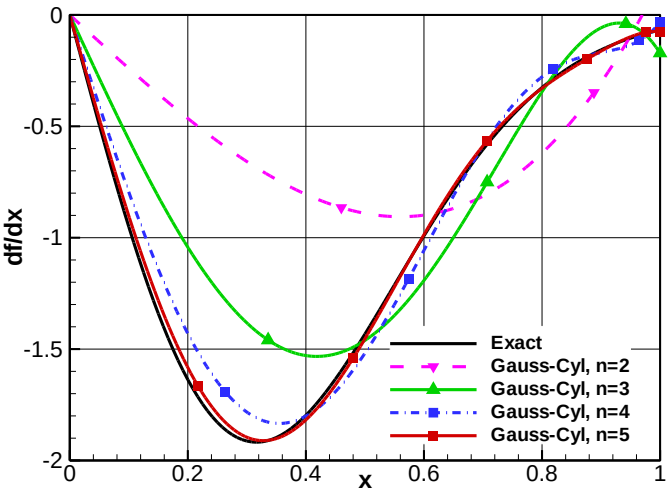

Fig. 2.8 Derivative approximation, cylindrical Gauss points

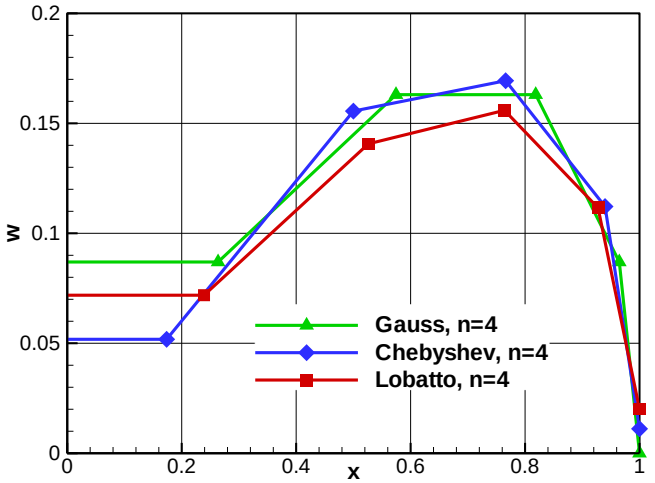

Fig. 2.9 Quadrature points and weights, cylindrical n = 4

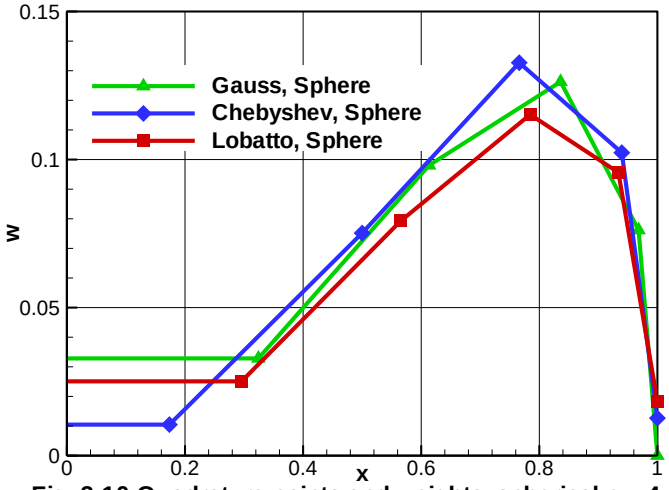

Fig. 2.10 Quadrature points and weights, spherical n = 4

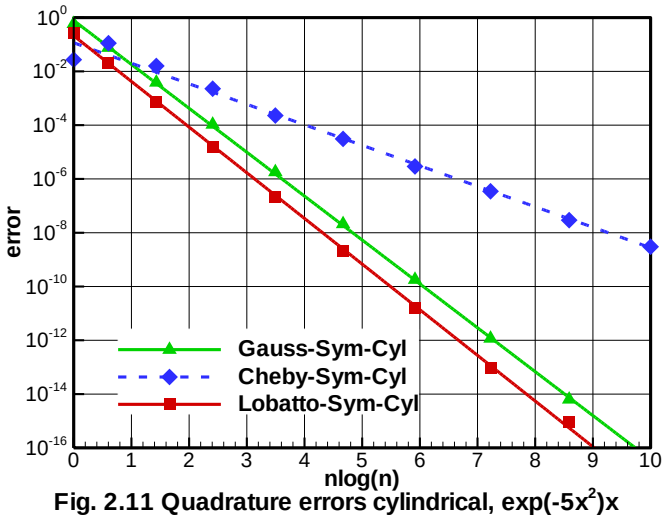

Fig. 2.11 Quadrature errors cylindrical, exp(-5x^2)x

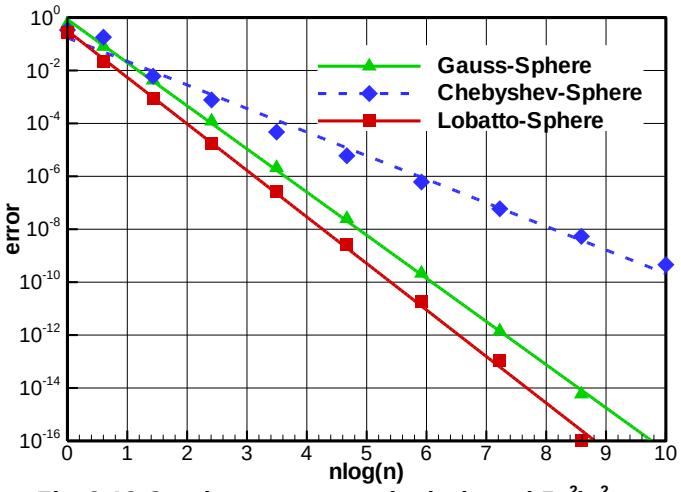

Fig. 2.12 Quadrature errors spherical, exp(-5x^2)x^2

The code above is for a symmetric problem, and some plots are shown with cylindrical and spherical geometry. First, Lagrange interpolating polynomials, `Li`, are shown for four cylindrical Gauss points in Fig. 2.6. Then the approximation of the exponential function, `f` and `fi`, and its first derivative, `df` and `dfa`, are shown for cylindrical Gauss points in Figs. 2.7 and 2.8. The quadrature weights, `w`, are shown in Figs. 2.9 and 2.10 for four points in cylindrical and spherical geometry, respectively. The error in the integrals from 0 to 1 of $exp(-5x^2)x$ and $exp(-5x^2)x^2$ are shown in Figs. 2.11 and 2.12, i.e. `(fx-fw)/fx` for cylindrical and spherical geometry.

When examining the Figs. 2.9 and 2.10, note that Chebyshev points are not shifted toward x = 1 as they are for the other points (due to β in the polynomial weight, see Eq. (2.6) and Table 2.1). The Chebyshev points are not optimal for integration, which is apparent in Figs. 2.11 and 2.12. The approximate integral converges with $(n)log(n)$, so for large n all give exponential convergence. However, the convergence rate is slower for Chebyshev points by about a factor of 2. Chebyshev points are optimal for interpolation, while Gauss and Lobatto points are optimal for integration.