An Analysis of Five Numerical Methods for Approximating Certain Hypergeometric Functions in Domains within their Radii of Convergence

John Pearson

MSc Special Topic

Abstract

Numerical approximations of the Bessel function $J_0(x)$ and the hypergeometric functions ${}_1F_1(0.1; 0.2; x)$ and ${}_2F_1(-0.9, 0.3; -0.2; x)$ are sought. Three methods for approximating these functions involve solving numerically the ordinary differential equations which they satisfy; these are the finite difference method, the shooting method and the use of a Chebyshev differentiation matrix. In addition, two methods from approximation theory, ℓ_2 approximation and the use of Padé approximants, are used to compute the Taylor series of the three functions. The ℓ_1, ℓ_2 and ℓ_∞ errors for each method are analysed for varying numbers of mesh points, as are the computation times relative to those for the inbuilt Matlab functions. The most effective methods for computing the three functions are proposed.

Contents

1 Introduction

Hypergeometric functions are a type of special functions that computer programs such as Matlab have trouble computing quickly. The purpose of this project is to use a number of numerical methods to establish approximations to the hypergeometric functions ${}_1F_1(a;c;x)$ and ${}_2F_1(a,b;c;x)$ for specially selected values of a, b and c in each case, in order to gain an idea of which, if any, of these methods are suitable for large-scale computations of the functions in general. In addition, the Bessel function, $J_0(x)$, which is closely related to the hypergeometric function, ${}_0F_1$, and can be computed much faster by Matlab, is computed in order to test the effectiveness of the methods before using them on ${}_1F_1$ and ${}_2F_1$.

Three of the methods which will be used to compute the functions above will attempt to solve boundary value problems which the functions satisfy. One of these methods is the finite difference method, which involves discretising the space in which the solution is to be found into a number of mesh points, approximating the derivatives in the differential equations in order to find a matrix system, and solving the system to find an approximate solution at each of the mesh points. Another method is the shooting method, in which the boundary value problem is formulated in terms of four first order initial value problems and then solved. The third method involves constructing Chebyshev differentiation matrices for the derivatives and solving that system.

Further, two methods are used to approximate the Taylor series of the functions of which an approximation is sought. ℓ_2 approximation is used to represent the series in terms of a low order polynomial, and Padé approximants are found in terms of a ratio of two polynomials of low order.

The times taken for each of the methods to compute the Bessel function and the hypergeometric functions compared to the inbuilt Matlab functions will then be analysed, as will the ℓ_1, ℓ_2 and ℓ_∞ errors, assuming, for the purpose of this investigation, the Matlab computation to be the correct one.

2 Background: Hypergeometric functions and their properties

Hypergeometric functions are a type of **special function**, the computation of which is frequently useful in mathematics, physics and other areas of science. As outlined in [1], for $p, q \in \mathbb{Z}$ and $z \in \mathbb{C}$, the function ${}_pF_q(a_1, ..., a_p; b_1, ..., b_q; x)$ is defined by:

$$ {}_pF_q(a_1, ..., a_p; b_1, ..., b_q; x) = \sum_{n=0}^{\infty} \frac{(a_1)_n ... (a_p)_n}{(b_1)_n ... (b_q)_n} \frac{z^n}{n!} \tag{2.1} $$

where, for some parameter ρ:

$$ (\rho)_n = \rho(\rho+1)...(\rho+n-1), \ (\rho)_0 = 1 \ . $$

A number of common functions are expressible as hypergeometric functions; for example:

$$ {}_0F_0(\ ;\ ;z) = e^z, \ {}_1F_0(-\omega;\ ;z) = (1-z)^{\omega}, \ z\ {}_2F_1\left(\frac{1}{2}, \frac{1}{2}; \frac{3}{2}; z\right) = \sin^{-1} z \ . $$

The derivative of the hypergeometric function, ${}_pF_q$, can be easily expressed as:

$$ \frac{\mathrm{d}}{\mathrm{d}z} \left({}_pF_q(a_1, ..., a_p; b_1, ..., b_q; z)\right) = \frac{a_1 ... a_p}{b_1 ... b_q} {}_pF_q(a_1 + 1, ..., a_p + 1; b_1 + 1, ..., b_q + 1; z) \ . $$

The n-th derivative can be expressed as:

$$\frac{\mathrm{d}^n}{\mathrm{d}z^n}\left({}_pF_q(a_1,...,a_p;b_1,...,b_q;z)\right) = \frac{(a_1)_n...(a_p)_n}{(b_1)_n...(b_q)_n}{}_pF_q(a_1+n,...,a_p+n;b_1+n,...,b_q+n;z) \quad (2.2)$$

An important property that we will need to use is the convergence criteria of the hypergeometric functions depending on the values of p and q. Provided a_i and b_i are not non-positive integers for any i, the relevant cases are, using the **ratio test**:

- If $p \leq q$, then the ratio of coefficients of z^n in the Taylor series of the hypergeometric function, ${}_pF_q$, tends to 0 as $n \to \infty$, and hence the radius of convergence is ∞, i.e. the series converges for all values of $|z|$. In particular, the radius of convergence for ${}_0F_1$ and ${}_1F_1$ is ∞.
- If $p = q + 1$, the ratio of coefficients of z^n tends to 1 as $n \to \infty$, so the radius of convergence is 1, i.e. the series converges only if $|z| < 1$. In particular ${}_2F_1$ converges only for $|z| < 1$.
- If $p > q + 1$, the ratio of coefficients of z^n tends to ∞ as $n \to \infty$, so the radius of convergence is 0, i.e. the series does not converge for any value of $|z|$.

The approximations to the relevant hypergeometric functions will only be sought within the radii of convergence. For $p = q + 1$, there is a further restriction for convergence on the unit circle; the series only converges absolutely at $z = 1$ if $\Re(\sum_{i=1}^{q} b_i - \sum_{i=1}^{p} a_i) > 0$, so the selection of values for a_i and b_i must reflect that.

Two of the most commonly used hypergeometric functions in practice are ${}_1F_1$ and ${}_2F_1$; it is these that will be computed using five numerical methods. It will be illustrative to test the methods on these functions, as the inbuilt Matlab function '**hypergeom**' takes a significant amount of time to compute them. However, it will be helpful to first use the methods on a function which takes less time for Matlab to compute, so that the code to implement the methods can be tested quickly and effectively. The function that will be used to do this is the **Bessel function**, $J_0(x)$, which is closely related to the **confluent hypergeometric limit function**, ${}_0F_1$. For α, a non-negative integer, and x, a real number (although the definition holds for complex values), $J_\alpha(x)$ is given by:

$$J_\alpha(x) = \sum_{i=0}^{\infty} \frac{(-1)^i}{i!\Gamma(i+\alpha-1)}\left(\frac{x}{2}\right)^{2i+\alpha} = \frac{\left(\frac{x}{2}\right)^\alpha}{\Gamma(\alpha+1)}{}_0F_1(\ ;\alpha+1;-\frac{x^2}{4}) \quad (2.3)$$

and satisfies the ordinary differential equation:

$$x^2\frac{\mathrm{d}^2J_\alpha}{\mathrm{d}x^2} + x\frac{\mathrm{d}J_\alpha}{\mathrm{d}x} + (x^2-\alpha^2)J_\alpha = 0\ . \quad (2.4)$$

The zeros of the function $J_0(x)$ are widely known; in particular the 4th zero is at $x = 11.79153444$ (to 10 significant figures), as is the value of $J_0(x)$ at $x = \pm 1$, which is 0.7651976866 at both points. This will be used in the computations.

By using the above expression for differentiation of terms of hypergeometric functions, we can see that the **confluent hypergeometric function**, ${}_1F_1(a;c;x)$, where x is a real-valued number, satisfies the differential equation:

$$x\frac{\mathrm{d}^2F}{\mathrm{d}x^2} + (c-x)\frac{\mathrm{d}F}{\mathrm{d}x} - aF = 0\ . \quad (2.5)$$

It can be shown that the general solution of the above differential equation is:

$$F = \lambda\ {}_1F_1(a;c;x) + \mu\ {}_1F_1(1+a-c;2-c;x)$$

where λ, μ are constants. Therefore, in this project, for the numerical methods for solving the differential equation to approximate ${}_1F_1$, a value $c < 1$ will be chosen to ensure that the second part of the solution, ${}_1F_1(1 + a - c; 2 - c; x)$, does not contribute to the solution. Hence, if appropriate boundary conditions are chosen, solving the differential equation will give the required approximation to ${}_1F_1$. The values for a and c chosen to test the methods of approximating this function will be $a = 0.1$, $c = 0.2$, and it will be approximated on the interval $[-1, 1]$.

The final function to be approximated, ${}_2F_1(a, b; c; x)$, where x takes real values, satisfies the differential equation:

$$x(1-x)\frac{\mathrm{d}^2F}{\mathrm{d}x^2} + [c - (a+b+1)x]\frac{\mathrm{d}F}{\mathrm{d}x} - abF = 0 \ . \tag{2.6}$$

The general solution here is:

$$F = \lambda \ {}_1F_1(a, b; c; x) + \mu \ {}_1F_1(1 + a - c, 1 + b - c; 2 - c; x) \ .$$

So again a value of $c < 1$ will be chosen to avoid the appearance of the second solution.

For the boundary conditions, two known formulae from [1] are used:

$$ {}_2F_1(a, b; c; 1) = \frac{\Gamma(c)\Gamma(c-a-b)}{\Gamma(c-a)\Gamma(c-b)} \tag{2.7}$$

$$ {}_2F_1(a, b; 1 + a - b; -1) = 2^{-a}\sqrt{\pi}\frac{\Gamma(1+a-b)}{\Gamma(1+\frac{1}{2}a-b)\Gamma(\frac{1}{2}+\frac{1}{2}a)} \ . \tag{2.8}$$

In order to make use of the second equation, the value $c = 1+a-b$ is chosen. Then, the assumption $c < 1$, mentioned above, is taken into account, and the convergence criterion $c > a + b$ (the inequality $\Re(\sum_{i=1}^{q} b_i - \sum_{i=1}^{p} a_i) > 0$ applied to ${}_2F_1$) used, in order to give the requirements:

$$a < b < \frac{1}{2}, \ c = 1 + a - b \ .$$

with a, b and c non-integers. The values chosen to test the numerical methods will be $a = -0.9$, $b = 0.3$, $c = -0.2$. In accordance with the convergence criteria above, the solution will be sought on the interval $-1 < x < 1$.

Graphs of the three functions to be approximated are shown overleaf in Figure 1.

The effectiveness of the numerical approximations made using the five methods will depend to a large extent on how far the approximations are from the correct solution, but there is an issue of how this is measured. For the purpose of this project, we assume that the numerical solution generated by the inbuilt Matlab function, 'hypergeom' (or 'besselj' where the Bessel function is being computed), is the correct one. Each of the methods used will involve taking an equally spaced set of points within the domain of approximation. Therefore, the difference between the computation of the function by Matlab and the numerical approximation using one of these five methods can be computed at each of these points and used to form a vector, which will be denoted $\mathbf{v} = [v_1, ..., v_{n+1}]^T$, where there are $n + 1$ points at which the solution is approximated.

The three measures which will be used to determine the accuracy of the approximation will be the ℓ_1, ℓ_2 and ℓ_∞ errors defined by:

$$\|\mathbf{v}\|_{\ell_1} = \sum_{i=1}^{n+1} |v_i|, \ \ \|\mathbf{v}\|_{\ell_2} = \left(\sum_{i=1}^{n+1} |v_i|^2\right)^{\frac{1}{2}}, \ \ \|\mathbf{v}\|_{\ell_\infty} = \max_i v_i \ . \tag{2.9}$$

A low value for the ℓ_∞ error ensures that the numerical approximation is accurate at every point at which it has been evaluated. Low values for ℓ_1 and ℓ_2 are likely to occur if the errors are small at a large number of points or are consistently low. Hence, all three errors are useful indications of how good the approximations are.

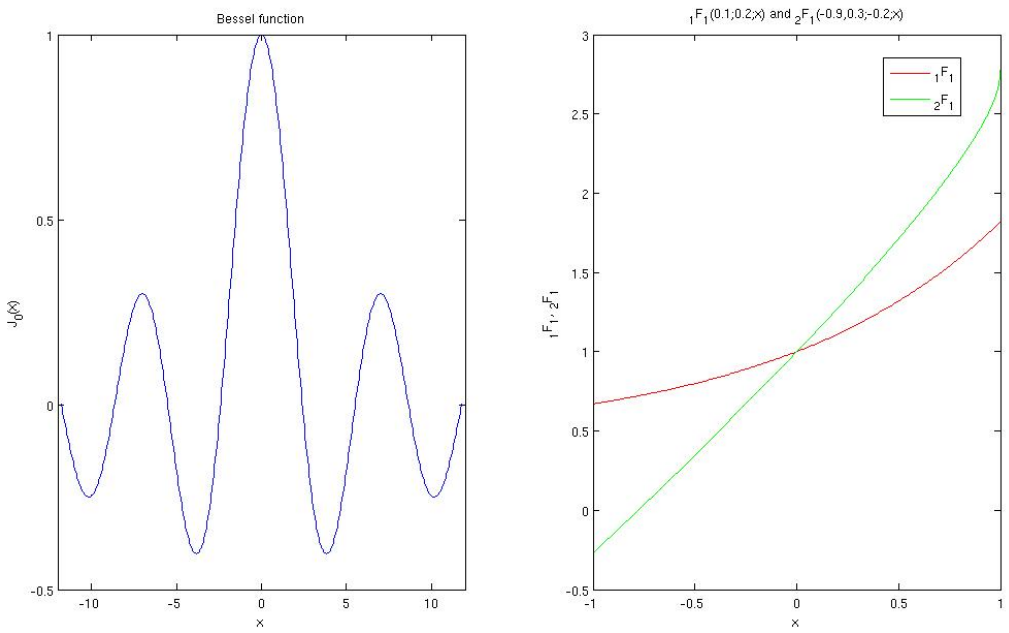

Figure 1: Graphs of the three functions for which an approximation is sought, $J_0(x)$, ${}_1F_1(0.1; 0.2; x)$ and ${}_2F_1(-0.9, 0.3; -0.2; x)$

3 Finite difference method

3.1 Fundamentals

The **finite difference method** is used to numerically solve ordinary differential equations of the form:

$$Lu = f(x) \text{ on } [x_{min}, x_{max}]$$

where Lu denotes a differential operator and boundary conditions are given at x_{min} and x_{max}. In this problem, the situation is more straightforward as we simply have Dirichlet boundary conditions at the boundary points.

In order to formulate the finite difference scheme, we need to create a mesh by discretising the domain of interest, which is the interval $[x_{min}$,$x_{max}]$. We do so by splitting the interval into m subintervals of equal length, bounded by the points $x_{min} = x_0$,x_1,x_2,...,x_{m-1},$x_m = x_{max}$. The step-size is defined to be $h = \frac{x_{max}-x_{min}}{m}$. The true solution at the point $x_i,\ i = 0, 1, ..., m$ is given by $u_i = u(x_i)$, and U_i denotes the approximate solution at that point.

The numerical approximation to the solution is made by approximating $u(x_i) \approx U_i$ and the derivatives of u at the mesh points. We can approximate $u'(x_i),\ i = 0, 1, ..., m$ by the implicit Euler method or the trapezium rule, which are respectively:

$$u'(x_i) \approx \frac{U_{i+1} - U_i}{h},\ u'(x_i) \approx \frac{U_{i+1} - U_{i-1}}{2h}\ .$$

We use the second of the above approximations as the roundoff error generated by the approximation is of order h^2 rather than of order h as in the first one, so taking the second approximation will have significant advantages. We approximate $u''(x_i)$ by:

$$u''(x_i) \approx \frac{U_{i+1} - 2U_i + U_{i-1}}{h^2},\ i = 0, 1, ..., m\ .$$

Hence, for the model second order ordinary differential equation $a(x)u''(x)+b(x)u'(x)+c(x)u(x)=f(x)$, applying the method of finite differences gives, for $i=1,2,...,m-1$:

$$L_hU_i \equiv a(x_i)\frac{U_{i+1}-2U_i+U_{i-1}}{h^2}+b(x_i)\frac{U_{i+1}-U_{i-1}}{2h}+c(x_i)U_i=f(x_i) \tag{3.1}$$

before we have taken account of boundary conditions, which will give us equations for U_0 and U_m. From this method, we will then have a system of $m+1$ simultaneous equations for the numerical approximations $U_0, U_1, ..., U_m$, which can be written as a matrix system $\mathbf{Au}=\mathbf{b}$. If we have Dirichlet boundary conditions at x_{min} and x_{max}, we will have:

$$\begin{aligned}
\mathbf{A} &= \begin{bmatrix}
1 & 0 & 0 & \cdots & \cdots & \cdots & 0 \\
\frac{a_0}{h^2}-\frac{b_0}{2h} & -\frac{2a_1}{h^2}+c_1 & \frac{a_2}{h^2}-\frac{b_2}{2h} & 0 & \cdots & \cdots & 0 \\
0 & \frac{a_1}{h^2}-\frac{b_1}{2h} & -\frac{2a_2}{h^2}+c_2 & \frac{a_3}{h^2}-\frac{b_3}{2h} & 0 & \cdots & 0 \\
\vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\
0 & 0 & \cdots & \cdots & \cdots & -\frac{2a_{m-1}}{h^2}+c_{m-1} & \frac{a_m}{h^2}-\frac{b_m}{2h} \\
0 & 0 & \cdots & \cdots & \cdots & 0 & 1
\end{bmatrix} \\
\mathbf{u} &= \begin{bmatrix} U_0 \\ U_1 \\ U_2 \\ \vdots \\ U_{m-1} \\ U_m \end{bmatrix}, \ \mathbf{b} = \begin{bmatrix} u(x_0) \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_{m-1}) \\ u(x_m) \end{bmatrix}
\end{aligned} \tag{3.2}$$

where $a_k=a(x_k)$, $b_k=b(x_k)$, $c_k=c(x_k)$.

The global error, e_i, at the point x_i is the difference between the true function value there and the numerical approximation:

$$e_i=u_i-U_i, \ i=0,1,...,m$$

and the truncation error T_i is given by:

$$T_i=L_hu_i-f_i=L_h(u_i-U_i)=L_he_i, \ i=0,1,...,m$$

where $u_i=u(x_i)$.

Now, all the ordinary differential equations we are trying to solve are of the form:

$$\alpha(x)u''(x)+\beta(x)u'(x)+\gamma(x)u(x)=0$$

with Dirichlet boundary conditions at the end-points. Therefore, $f_i=0$ for all i, and to find L_hu_i we need to Taylor expand u_{i+1} and u_{i-1} as follows:

$$u_{i+1}=u_i+hu'(x_i)+\frac{h^2}{2!}u''(x_i)+\frac{h^3}{3!}u'''(x_i)+O(h^4), \ u_{i-1}=u_i-hu'(x_i)+\frac{h^2}{2!}u''(x_i)-\frac{h^3}{3!}u'''(x_i)+O(h^4)$$

so that when we substitute these values into the expression for T_i, using the fact that $f(x)=0$ for all three equations which will be considered, we obtain:

$$\begin{aligned}
T_i=L_hu_i &= \frac{a(x_i)}{h^2}[u_{i+1}-2u_i+u_{i-1}]+\frac{b(x_i)}{2h}[u_{i+1}-u_{i-1}]+c(x_i)u_i-\underbrace{f_i}_{0} \\
&= \left[\frac{a(x_i)}{12}u^{(iv)}(x_i)+\frac{b(x_i)}{6}u'''(x_i)\right]h^2+O(h^4) \ .
\end{aligned} \tag{3.3}$$

So we therefore have that $\|T\|_\infty = \overset{max}{i}\ T_i = O(h^2)$, and hence:

$$T_i = L_h e_i \Rightarrow e_i = L_h^{-1} T_i \Rightarrow \|e\|_\infty \leq \left\|L_h^{-1}\right\|_\infty \|T\|_\infty$$

using the property of the norm of the product of two matrices, where $\left\|L_h^{-1}\right\|_\infty$ is the **spectral radius**, or largest eigenvalue, of L_h^{-1}. We would therefore expect the global error in the ℓ_∞ norm to be of order h^2.

We will examine not only these ℓ_∞ errors, but also ℓ_1 and ℓ_2 errors, as well as comparing computational times with those for the inbuilt Matlab function for computing the hypergeometric functions.

3.2 Brief outline of code used

In order to instruct Matlab to calculate a numerical approximation to the differential equations using the finite difference method, we need to include the following steps in the code:

- Define the range of x over which we wish to solve the differential equation, the number of points we wish to approximate it at, $m+1$, and the step size, h.
- Construct an $(m+1)$-vector $\mathbf{u} = [U_0, U_1, ..., U_m]^T$ of numerical approximations, in terms of which we will set up an algebraic system of equations.
- Construct $(m+1)\times(m+1)$ differentiation matrices corresponding to differentiating u once (to find u') and twice (to find u''). Also construct a matrix $\mathbf{I}_{m+1}$ corresponding to the insertion of a u term in the equation.
- Multiply each entry in these matrices by the terms that u, u' and u'' are multiplied by in the equation. Sum these matrices to obtain a finite differences matrix, $\mathbf{A}$, so that we have fully discretised the ordinary differential equation. Incorporate the boundary terms by making the first and last rows zero apart from the first entry of the first row and the last entry of the last row which we make 1.
- Create an $m+1$-vector $\mathbf{b}$ of zeros (corresponding to the fact that $f(x) = 0$ in the ordinary differential equations). Then overwrite the first and last entries by the known boundary values of u.
- Solve the system $\mathbf{Au} = \mathbf{b}$ to find the vector of numerical approximations at each mesh point.

The code used is shown in Appendix A.

3.3 Numerical results

Below is a table comparing the time taken to compute the Bessel function, $J_0(x)$, using the finite difference method with that using the function '**besselj**' in Matlab (the code was run three times and the time taken for each averaged to give a better idea of the time typically taken), as well as the ℓ_1, ℓ_2 and ℓ_∞ error in the finite difference results, assuming the 'besselj' function gives the exact answer. The results are given for different numbers of mesh points n on $[-11.79153444, 11.79153444]$.

n	Time taken	Time taken with 'besselj'	ℓ_1	ℓ_2	ℓ_∞
20	0.000477	0.000460	13.1484	3.7778	1.9221
50	0.000579	0.000632	3.7330	0.6726	0.2237
100	0.000897	0.000925	0.9295	0.1652	0.05446
200	0.002378	0.001455	0.5249	0.06569	0.01543
500	0.011609	0.003033	0.2512	0.02811	0.002971
1000	0.063335	0.006352	0.1423	0.007975	8.472e-4
2000	0.232652	0.009113	0.07967	0.003164	2.388e-4
5000	0.968251	0.011905	0.07284	0.001297	4.406e-5

We can see that, considering the length of the interval being integrated over and the relatively large change occurring within the Bessel function within this region, the results for the errors are reasonable. The ℓ_∞ error seems to decrease roughly by a factor of 4 when the step size is halved, as expected.

The fact that Matlab is very efficient at evaluating the Bessel function means that the finite difference method is not particularly useful here, as it takes longer to solve the differential equation. However, the reasonable accuracy obtained gave encouraging signs for when the hypergeometric functions ${}_1F_1$ and ${}_2F_1$ were to be computed. Below is a graph showing the exact and numerical solutions (which overlap due to how close the approximation is to the exact solution), and a graph showing the absolute error at each point where the approximation was made:

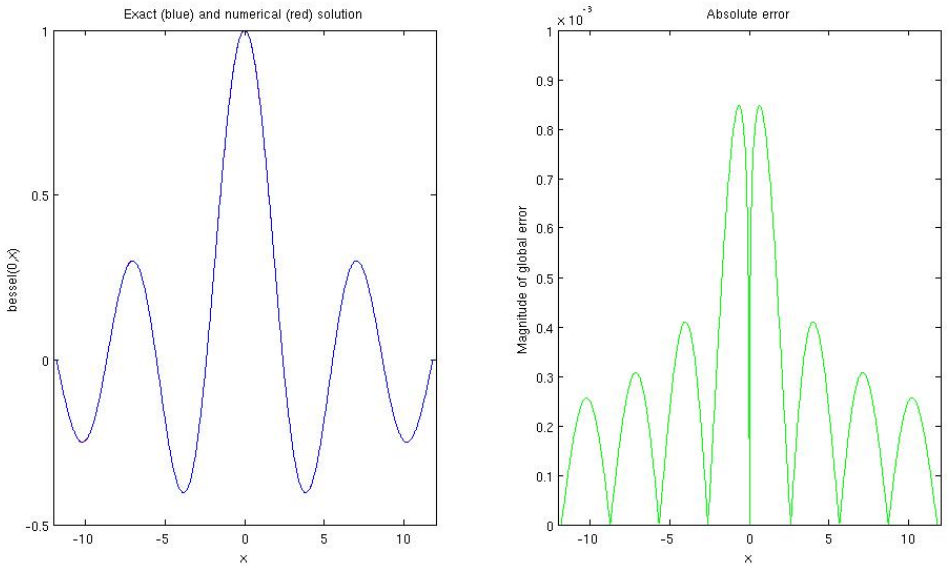

Figure 2: Numerical results for the finite difference method applied to $J_0(x)$ with 500 mesh points (the approximation overlaps with the true solution due to the good accuracy)

Below is a table showing a comparison of times taken for the finite difference method and the Matlab function '**hypergeom**' to compute the function ${}_1F_1(0.1; 0.2; x)$ on $[-1, 1]$, together with the various errors.

n	Time taken	Time taken with 'hypergeom'	ℓ_1	ℓ_2	ℓ_∞
10	0.000397	0.061790	0.003504	0.001355	7.966e-4
20	0.000643	0.104515	0.001643	4.307e-4	1.708e-4
50	0.000938	0.896300	6.313e-4	1.027e-4	2.537e-5
100	0.002969	1.637009	3.113e-4	3.560e-5	6.194e-6
200	0.005874	2.813293	1.546e-4	1.247e-5	1.530e-6
500	0.062331	5.072927	6.518e-5	3.137e-5	2.431e-7
1000	0.259691	9.628050	3.075e-5	1.107e-5	6.663e-8

This table shows that the finite difference method used has impressive ℓ_1, ℓ_2 and ℓ_∞ error properties. As n increases, the ℓ_1 and ℓ_2 errors decrease, even though the vectors which they are measuring are becoming longer, and for large n the ℓ_∞ error is of the order of 10^{-8}. Furthermore, the time taken is orders of magnitude smaller than that taken by the function 'hypergeom' in Matlab, which suggests that this method gives a successful approximation.

Below is a table showing the corresponding results for ${}_2F_1(-0.9, 0.3; -0.2; x)$ on $[-1, 1]$, as well as the errors solely for the first half of the vectors, i.e. for $[-1, 0]$.

n	Time taken	'hypergeom' time	ℓ_1	ℓ_2	ℓ_∞	ℓ_1 on $[-1,0]$	ℓ_2 on $[-1,0]$	ℓ_∞ on $[-1,0]$
10	0.000423	0.367939	0.5980	0.2395	0.1352	1.930e-4	7.186e-5	3.301e-5
20	0.000817	0.562589	1.0136	0.2751	0.1085	9.633e-5	2.437e-5	7.703e-6
50	0.0001508	1.673386	1.8619	0.3124	0.07750	3.825e-5	6.002e-6	1.181e-6
100	0.003034	2.901258	2.8717	0.3382	0.05956	1.905e-5	2.102e-6	2.909e-7
200	0.010952	4.421957	4.3870	0.3641	0.04569	9.504e-6	7.393e-7	7.217e-8
500	0.035431	7.365639	7.6322	0.3998	0.03197	3.795e-6	1.864e-7	1.149e-8
1000	0.139030	14.925628	11.5787	0.4286	0.02436	1.896e-6	6.581e-8	2.867e-9

Below is a graph showing the results for the computation of ${}_2F_1$:

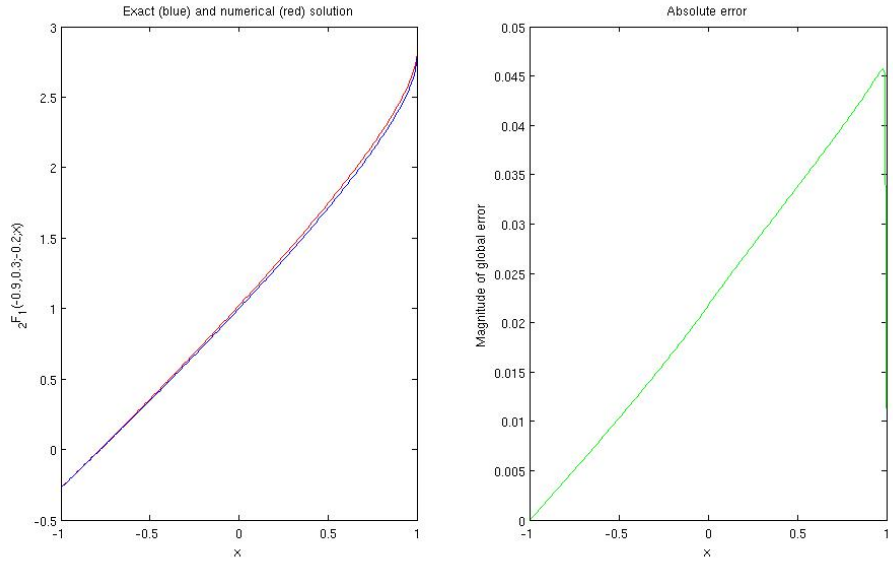

Figure 3: Numerical results for the finite difference method applied to ${}_2F_1$ with 200 mesh points

The results for the errors over the entire interval, and in particular that the ℓ_∞ error does not decrease with h^2, suggest that this method is ineffective at approximating ${}_2F_1$. However, the graph in Figure 3 shows that the error is largely on the positive x-values, a point that can also be seen by the errors in the table for the interval $[-1, 0]$. This suggests that the problem with approximating the ${}_2F_1$ function is the fact that there are two singular points in the interval of integration, one of which is in the middle of the region. This seems to have a knock-on effect when the differentiation matrices are computed. Hence, it is reasonable to suppose that, to compute ${}_2F_1$ over the entire interval $[-1, 1]$, a more sophisticated method is required.

4 Shooting method

4.1 Fundamentals

The idea of the **shooting method** is to reformulate the problem of finding the numerical solution of a second order ordinary differential equation with boundary conditions, of the form:

$$Lu = f(x) \text{ on } [x_{min}, x_{max}],\ u(x_{min}) = u_{min},\ u(x_{max}) = u_{max}$$

as a problem of finding the numerical solution of two initial value problems instead. So suppose we have the problem:

$$u''(x) + \xi(x)u'(x) + \eta(x)u(x) = g(x),\ u(x_{min}) = u_{min},\ u(x_{max}) = u_{max}\ . \tag{4.1}$$

We define $u(x; s)$ to be the solution to:

$$u''(x) + \xi(x)u'(x) + \eta(x)u(x) = g(x),\ u(x_{min}) = u_{min},\ u'(x_{min}) = s\ .$$

Now, let $u_0 = u(x; 0)$, so:

$$u_0''(x) + \xi(x)u_0'(x) + \eta(x)u_0(x) = g(x),\ u_0(x_{min}) = u_{min},\ u_0'(x_{min}) = 0\ .$$

Now, suppose that $u = u_0 + \mu u_1$, where μ is a constant, and u_1 satisfies:

$$u_1''(x) + \xi(x)u_1'(x) + \eta(x)u_1(x) = 0,\ u_1(x_{min}) = 0,\ u_1'(x_{min}) = 1\ .$$

Then it is clear that u must satisfy the first boundary condition $u(x_{min}) = u_{min}$. For the second boundary condition to be satisfied, we need:

$$u(x_{max}) = u_0(x_{max}) + \mu u_1(x_{max}) = \beta\ .$$

So, provided we can find exact solutions to the two initial value problems:

$$u_0''(x) + \xi(x)u_0'(x) + \eta(x)u_0(x) = g(x),\ u_0(x_{min}) = u_{min},\ u_0'(x_{min}) = 0 \tag{4.2}$$

$$u_1''(x) + \xi(x)u_1'(x) + \eta(x)u_1(x) = 0,\quad u_1(x_{min}) = 0,\quad u_1'(x_{min}) = 1 \tag{4.3}$$

then $u(x) = u_0(x) + \frac{u_{max} - u_0(x_{max})}{u_1(x_{max})} u_1(x)$ will solve exactly:

$$u''(x) + \xi(x)u'(x) + \eta(x)u(x) = g(x),\ u(x_{min}) = u_{min},\ u(x_{max}) = u_{max}\ .$$

We now need to consider a method for solving the initial value problems in Equations (4.2) and (4.3). So, substitutions will be made in order to convert each of the two equations into first order initial value problems, and then solve these. The substitutions

come by introducing new variables v_0 and v_1 as follows to form a set of four first order initial value problems:

$$\begin{aligned} u_0'(x) &= v_0(x) &&, \ u_0(x_{min}) = u_{min} && (4.4)\\ v_0'(x) &= -\xi(x)v_0 - \eta(x)u_0(x) + g(x), &&\ v_0(x_{min}) = 0 && (4.5)\\ u_1'(x) &= v_1(x) &&, \ u_0(x_{min}) = 0 && (4.6)\\ v_1'(x) &= -\xi(x)v_1 - \eta(x)u_1(x) + g(x), &&\ v_1(x_{min}) = 1 \ . && (4.7) \end{aligned}$$

It should be noted that the top two initial value problems are the same equations as the bottom two, but with different initial conditions.

As the problem has been reformulated, a method is now required to solve this new form. The method that will be used to solve each of the above four initial value problems is the **Runge-Kutta RK4** method, which, for the solution of the problem $y' = f(x, y)$, with $y(x_0)$ given, reads:

$$\begin{aligned} y_0 &= y(x_0) \\ y_{n+1} &= y_n + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} \qquad (4.8)$$

where:

$$\begin{aligned} k_1 &= f(x_n, y_n) \\ k_2 &= f(x_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_1) \\ k_3 &= f(x_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_2) \\ k_4 &= f(x_n + h, y_n + hk_3) \ . \end{aligned}$$

If for each pair of equations, we alternate between calculating $k_j, \ j = 1, 2, 3, 4$ for each equation and then calculate the next numerical approximation for each equation, we have an explicit numerical algorithm which can be used to solve the equations numerically.

As the shooting method gives perfect accuracy for linear problems up to the roundoff error that will arise from Matlab summing the computations, which is negligible as we will be dealing with errors of order 10^{-15} here, the accuracy of the method will depend solely on the performance of the RK4 method. By evaluating Taylor series expansions of k_1, k_2, k_3 and k_4, and substituting these into the expression for the truncation error:

$$T_i = \frac{y(x_{i+1}) - y(x_n)}{h} - \frac{1}{6}\left(k_1(x_i, y(x_i)) + 2k_2(x_i, y(x_i)) + 2k_3(x_i, y(x_i)) + k_4(x_i, y(x_i))\right)$$

we find that the RK4 method has $O(h^4)$ truncation error, and as $\|e\|_\infty \leq \left\|L_h^{-1}\right\|_\infty \|T\|_\infty$ as shown earlier it will have $O(h^4)$ roundoff error also. This will hopefully give desirable error properties when we form the approximate solution.

4.2 The issue of singular points and how it is resolved

Rewriting the second order initial value problems as two first order initial value problems, Equations (4.2) and (4.3) creates a difficulty however. In order to write the equations as shown, dividing by a certain factor is involved, either x in the case of the Bessel function and ${}_1F_1$, or $x(1-x)$ in the case of ${}_2F_1$. This results in singular points in the equations causing problems ($x = 0$ in the case of Bessel and ${}_1F_1$ and $x = 0, 1$ in the case of ${}_2F_1$). These singular points are all in the region in which the numerical solution is sought, so

when the RK4 method is applied, problems arise when carrying out the method at the singular point and thereafter. Hence, the shooting method cannot be used on its own.

The solution to this difficulty is to divide the region into more than one part, and to apply the shooting method in each part with appropriate substitutions. To illustrate how this is carried out, consider the equation for ${}_1F_1(a;c;x)$, where a, c are 0.1, 0.2 respectively:

$$xu'' + (c-x)u' - au = 0,\ u(-1) = \alpha,\ u(1) = \beta\ .$$

It is also known that $u(0) = 1$, by expanding the Taylor series for ${}_1F_1$ for any a, c and substituting in $x = 0$. Hence, the region $[-1, 1]$, in which the solution is sought, is divided up into two regions, $[-1, 0]$ and $[0, 1]$. The solution in the region $[-1, 0]$ can be found using the shooting method as described previously, as the singular point $x = 0$ due to the factor of x before the u'' term, is only encountered at the end of the region of integration. To solve in the region $[0, 1]$, the substitution $x = 1 - X$ is made, which, using the chain rule, converts the problem on $[0, 1]$ to:

$$(1-X)u''(X) - (c+X-1)u'(X) - au(X) = 0,\ u(X=0) = u(x=1) = \beta,\ u(X=1) = u(x=0) = 1\ .$$

The u'' coefficient is now $1-X$, so if this equation is integrated from $X = 0$ to $X = 1$, i.e. backwards from $x = 1$ to $x = 0$, there is no longer a singular point. The solution to this equation may now be found using the shooting method. After that, the numerical solution of u is converted from X to x coordinates, and hence a solution for u is known on each of the x-intervals $[-1, 0]$ and $[0, 1]$. The solution vectors for each of the intervals are then joined up, and so the complete numerical solution is known.

A similar principle is used for the equation satisfied by the Bessel function $J_0(x)$, except as the solution is wanted on $[-11.79153444, 11.79153444]$, the substitution $x = 11.791534444 - X$ is made to find the solution in the second interval instead.

However, there is a further problem when the solution for the ${}_2F_1(-0.9, 0.3; -0.2; x)$ function is sought, because if the appropriate substitution $x = 1 - X$ is made, the coefficient of u'' in the differential equation, this time $x(1-x)$, remains unaltered. Hence in this case, the solution needs to be found in three regions, $[-1, 0]$, $[0, \gamma]$ and $[\gamma, 1]$, where $0 < \gamma < 1$. Experimenting with the value of γ shows that a value of γ close to $\frac{1}{2}$ is ideal. Taking $\gamma = \frac{1}{2}$, a similar method may now be applied. The solution on $[-1, 0]$ may be found in the normal way using the shooting method. Substituting $x = \frac{1}{2} - X$ and applying the shooting method yields a solution on the interval $[0, \frac{1}{2}]$, and the further substitution $x = \frac{1}{2} + Y$ can be used in a similar way to find a solution on $[\frac{1}{2}, 1]$. These three solutions can then be combined to find a complete numerical approximation to ${}_2F_1$ on $[-1, 1]$.

4.3 Brief outline of code used

In order to implement the shooting method, the code which instructs Matlab must contain the following:

- State the original second order differential equation for the boundary value problem we wish to solve. Then define new parameters, equivalent to the definition of v_0 and v_1 above, in order to reformulate the second order boundary value problem as four first order initial value problems.

- Define the initial conditions for the four problems and define the number of interpolation points, $m + 1$, and hence the step-size h, that will be used.

- Create a loop to compute values of k_j at each mesh point, and then advance the numerical approximation for u_0 and v_0 for the first pair of equations, and u_1 and v_1 for the second pair of equations. This loop will take the form:

```
for i = 1:m
  k1u=f1(x(i),u(i),v(i));
  k1v=f2(x(i),u(i),v(i));
  k2u=f1(x(i)+0.5*h,u(i)+0.5*k1u*h,v(i)+0.5*k1v*h);
  k2v=f2(x(i)+0.5*h,u(i)+0.5*k1u*h,v(i)+0.5*k1v*h);
  k3u=f1(x(i)+0.5*h,u(i)+0.5*k2u*h,v(i)+0.5*k2v*h);
  k3v=f2(x(i)+0.5*h,u(i)+0.5*k2u*h,v(i)+0.5*k2v*h);
  k4u=f1(x(i)+h,u(i)+k3u*h,v(i)+k3v*h);
  k4v=f2(x(i)+h,u(i)+k3u*h,v(i)+k3v*h);
  u(i+1)=u(i)+h/6*(k1u+2*k2u+2*k3u+k4u);
  v(i+1)=v(i)+h/6*(k1v+2*k2v+2*k3v+k4v);
end
```

  This needs to be done for each pair of initial value problems.

- Once we have found the numerical approximations to the two second order initial value problems for u_0 and u_1 by solving the four first order initial value problems for u_0, v_0, u_1, v_1, we sum them, as $u(x) = u_0(x) + \frac{u_{max}-u_0(x_{max})}{u_1(x_{max})}u_1(x)$, to find the numerical solution to the original boundary value problem.

Taking the above steps can be used to find the numerical solution for negative x. Making appropriate substitutions, as explained in 4.2, can be used to find solutions in the remainder of the region in which we seek the solution. The solution vectors can then be joined up to find a numerical solution on the entire interval.

The code used is in Appendix B.

4.4 Numerical results

This is a table showing the ℓ_1, ℓ_2 and ℓ_∞ errors of the numerical approximation of $J_0(x)$ using the shooting method compared to the inbuilt Matlab function, 'besselj', as well as the comparative times for the two programs to be run, for different numbers of mesh points n:

n	Time taken	Time taken with 'besselj'	ℓ_1	ℓ_2	ℓ_∞
21	0.006761	0.000460	0.7496	0.01978	0.1173
51	0.023348	0.000632	0.05715	0.01139	0.004054
101	0.048628	0.000925	0.01173	0.001563	3.832e-4
201	0.094737	0.001455	0.001936	1.789e-4	3.093e-5
501	0.221590	0.003003	1.574e-4	9.119e-6	9.960e-7
1001	0.315548	0.006352	2.152e-5	8.811e-7	6.826e-8
2001	0.592601	0.009113	7.700e-6	1.352e-7	5.574e-8
5001	0.926392	0.011905	6.910e-6	1.221e-7	4.118e-9

The shooting method being used in conjunction with the RK4 method for intial value problems seems to have very favourable error properties compared with the finite difference method. This is not surprising, as the ℓ_∞ error seems to decrease roughly in proportion to h^4 as predicted by truncation error analysis on RK4. The ℓ_1 and ℓ_2 errors also seem to be substantially lower than for finite differences with a comparable number of mesh points. Again however, the inbuilt Matlab function takes a substantially shorter period of time to carry out the computation.

The numerical results for ${}_1F_1$ are shown below:

n	Time taken	Time taken with 'hypergeom'	ℓ_1	ℓ_2	ℓ_∞
11	0.009783	0.061790	5.782e-6	2.263e-6	1.396e-6
21	0.012292	0.104515	2.040e-6	6.321e-7	3.021e-7
51	0.022197	0.896300	4.188e-7	7.169e-8	1.975e-8
101	0.029453	1.637009	1.137e-7	1.348e-8	2.512e-9
201	0.052792	2.813293	2.963e-8	2.474e-9	3.203e-10
501	0.280815	5.072927	4.864e-9	2.570e-10	2.097e-11
1001	0.58112	9.628050	1.227e-9	4.590e-11	2.661e-12

Again, the ℓ_1, ℓ_2 and ℓ_∞ errors compare very favourably with the finite difference method; for example, all three errors with 50 mesh points with the shooting method used are lower than the errors generated by using 1000 points in the finite difference method, albeit with a longer computation time. The errors for large values of n are impressively small.

Below is a graph showing the numerical solutions for $u_0(x)$ (labelled 1st iteration) and $u_1(x)$ (labelled 2nd iteration) on each region where the shooting method was applied ($[-1, 0]$ and $[0, 1]$), the shooting method approximation obtained from them and its comparison with the exact solution according to 'hypergeom', and a graph of the true solution less the numerical solution at all mesh points:

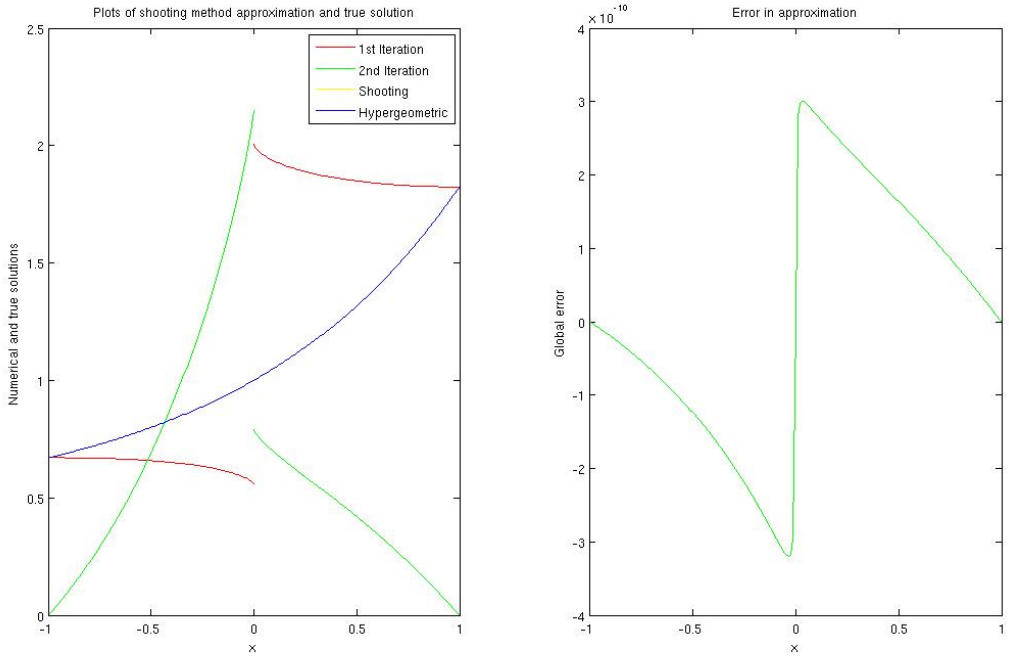

Figure 4: Numerical results for the shooting method applied to ${}_1F_1$ with 201 mesh points

The following table shows the results for the computation of ${}_2F_1$. It shows excellent accuracy on the interval $[-1, 0]$ but poor approximation close to $x = 1$.

n	Time taken	'hypergeom' time	ℓ_1	ℓ_2	ℓ_∞	ℓ_1 on $[-1,0]$	ℓ_2 on $[-1,0]$	ℓ_∞ on $[-1,0]$
21	0.016126	0.562589	0.1280	0.06637	0.04922	4.002e-7	1.371e-7	5.844e-8
49	0.045283	1.673386	0.2236	0.07760	0.04311	4.629e-8	1.042e-8	2.979e-9
101	0.086118	2.901258	0.3494	0.08539	0.03585	7.527e-9	1.189e-9	2.425e-10
201	0.151527	4.421957	0.5282	0.09205	0.02904	1.450e-9	1.640e-10	2.463e-11
501	0.250858	7.365639	0.9081	0.1007	0.02129	1.871e-10	1.356e-11	1.375e-12
1001	0.529782	14.925628	1.3678	0.1074	0.01658	4.270e-11	2.201e-12	1.648e-13

5 Chebyshev differentiation matrix

5.1 Principle of Chebyshev differentiation matrices

In many problems, using regularly spaced points does not give as accurate a numerical solution to a differential equation as possible. One alternative approach is to use **Chebyshev points**. If we wish to find n Chebyshev points on the interval $[-1, 1]$, they are given by:

$$-\cos\left(\frac{k\pi}{n-1}\right) \quad k = 0, 1, ..., n-1.$$

For the next method, we will expand on this idea in order to create a **Chebyshev differentiation matrix**, which will hopefully serve to approximate well the action of differentiating a function $u(x)$, much as the finite difference differentiation matrix did in Section 3. Having found this matrix, we can square it in order to find an approximation to differentiating the function $u(x)$ twice. As mentioned in [2], this approach will not work if the problem is **ill-conditioned** (which a matrix $\mathbf{A}$ is said to be if its **condition number** $\|\mathbf{A}\| \cdot \|\mathbf{A}^{-1}\|$ is large) or if the boundary conditions are not taken into account. However we will be taking the boundary conditions into account when we instruct Matlab to solve a matrix system, much as for the finite difference method.

One major difference between the finite difference matrix system and this one is that this one will not be sparse; in general, the matrix will be very dense. This will result in more computational work and hence a longer time taken to compute the solution. However, to compensate for this, it is hoped that the solution computed will be far more accurate.

In order to create the differentiation matrix, we will create an $m \times m$ matrix $\mathbf{M_1}$, for which the k-th column is equal to the **Chebyshev polynomial**, $T_{k-1}(x)$ for $-1 \leq x \leq 1$, where:

$$T_n(x) = \cos(n \cos^{-1}(x)) \tag{5.1}$$

which satisfies the recurrence relation:

$$T_0(x) = 1,\ T_1(x) = x,\ T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x),\ n = 1, 2, \ldots\ . \tag{5.2}$$

We will also create another $m \times m$ matrix, $\mathbf{M}_2$, whose k-th column is the derivative of $T_{k-1}(x)$, where:

$$T_n'(x) = \frac{n \sin(n \cos^{-1}(x))}{\sqrt{1-x^2}}, \quad -1 < x < 1 \tag{5.3}$$

$$T_n'(-1) = (-1)^{n+1} n^2 \tag{5.4}$$

$$T_n'(1) = n^2 \tag{5.5}$$

so the first column of $\mathbf{M_2}$ will be equivalent to the derivative of $T_n(x)$ at $x = -1$, the last column equivalent to $x = 1$, and the other entries determined using Equation (5.3).

We then create another matrix $\mathbf{D}$ which satisfies $\mathbf{DM_1}$=$\mathbf{M_2}$, in other words if we apply the matrix $\mathbf{D}$ to the system of Chebyshev polynomials evaluated at the Chebyshev points, we will obtain an approximation of the derivatives at those points.

We can then evaluate $\mathbf{D}^2$ to arrive at an approximation of a second derivative matrix; we will then substitute the two differentiation matrices into the differential equations for the hypergeometric functions to obtain approximate solutions.

5.2 Brief outline of Matlab code used

The code used for this method has the same basic structure as the finite differences code, but with different differentiation matrices used. In the code are the following instructions:

- Define the number of points $m+1$ we wish to use to approximate $u(x)$ at.
- Construct the points at which we wish to approximate the solution as the Chebyshev points in the interval $[-1, 1]$. Define an $m+1$-vector $\mathbf{u} = [U_0, U_1, ..., U_m]^T$ of numerical approximations at each of these Chebyshev points, in terms of which we will set up an algebraic system of equations.
- Construct $(m+1) \times (m+1)$ matrices corresponding to the Chebyshev polynomials and their derivatives as explained earlier. This code takes the form:

```
%Initialise matrices M1 and M2
  M1 = zeros(n,n);
  M2 = M1;

  for k = 0:n-1
%Define M1 in terms of Chebyshev polynomials,
%evaluated at the Chebyshev points
      M1(:,k+1) = cos(acos(x)*k);
%Define M2 in terms of the derivatives of the Chebyshev polynomials,
%taking care with M2(1,k+1) and M2(end,k+1),
%which correspond to the derivatives at x=-1 and x=1
      M2(:,k+1) = k*sin(acos(x)*k)./sqrt(1-x.^2);
      M2(1,k+1) = (-1)^(k+1)*k^2;
      M2(end,k+1) = k^2;
  end

%Solve a matrix system to obtain the differentiation matrix
  D = M2/M1;
```

  Square this differentiation matrix to obtain a matrix representing twice-differentiating.

- Multiply each entry in these matrices by the terms that u, u' and u'' are multiplied by in the equation. Sum these matrices to obtain a matrix, $\mathbf{A}$, so that we have fully discretised the ordinary differential equation. Incorporate the boundary terms by making the first and last rows zero apart from the first entry of the first row and the last entry of the last row which we make 1.
- Create an $m+1$-vector $\mathbf{b}$ of zeros (corresponding to the fact that $f(x) = 0$ in the ordinary differential equations). Then overwrite the first and last entries by the known boundary values of u.
- Solve the system $\mathbf{Au} = \mathbf{b}$ to find the vector of numerical approximations at each mesh point.

The code used is shown in Appendix C.

5.3 Numerical results

Below are two tables showing the numerical results for the approximation of $J_0(x)$ and ${}_1F_1(0.1; 0.2; x)$ on $[-1, 1]$, and a graph of the results for ${}_1F_1$:

n	Time taken	Time taken with 'besselj'	ℓ_1	ℓ_2	ℓ_∞
10	0.002007	0.000325	2.797e-9	1.308e-9	8.575e-10
20	0.002956	0.000460	2.416e-12	7.632e-13	3.347e-13
30	0.004276	0.000512	6.865e-12	2.150e-12	1.123e-12
50	0.007793	0.000632	2.383e-10	4.851e-11	1.332e-11
100	0.023438	0.000925	2.259e-9	3.460e-10	6.637e-11
200	0.098532	0.001455	7.093e-8	8.202e-9	1.507e-9

n	Time taken	Time taken with 'hypergeom'	ℓ_1	ℓ_2	ℓ_∞
10	0.057926	0.061790	3.205e-9	1.348e-9	7.786e-10
20	0.068239	0.104515	8.403e-12	2.797e-12	1.130e-12
30	0.087233	0.309782	1.866e-11	5.164e-12	1.768e-12
50	0.119155	0.896300	9.028e-10	1.456e-10	2.796e-11
100	0.146690	1.637009	3.358e-9	4.908e-10	8.736e-11
200	0.168721	2.813293	1.052e-7	1.003e-8	1.547e-9

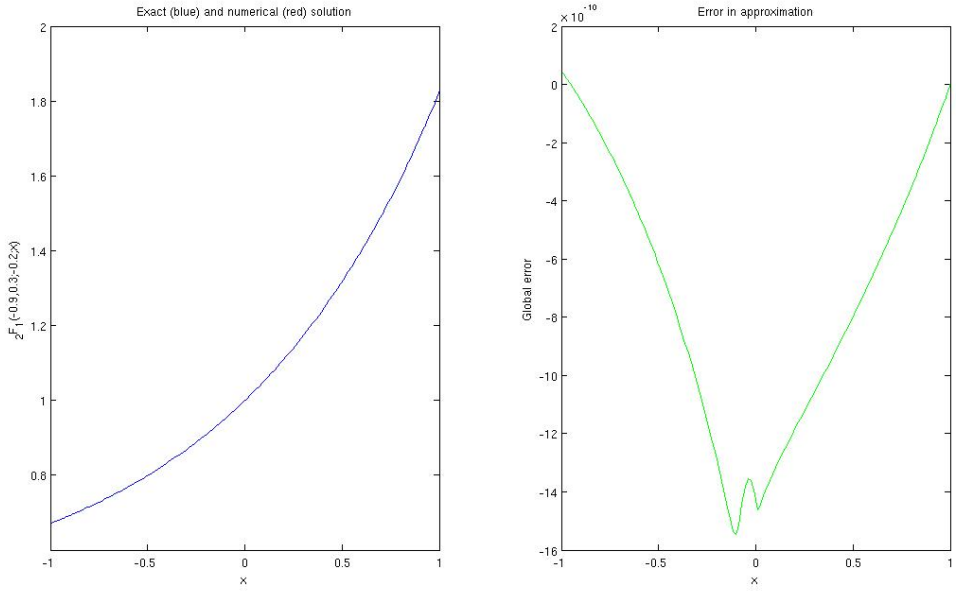

Figure 5: Exact and numerical solution for the Chebyshev differentiation matrix applied to ${}_1F_1$ with 200 mesh points (which overlap) and the error at each mesh point

The times taken for the computations involving Chebyshev differentiation matrices are much greater than those taken for the finite difference method and shooting method, because the differentiation matrices are dense (as opposed to the sparse matrices involved in the finite difference method), and hence the system of equations takes a substantial amount of time for the computer to solve. However, the times taken for ${}_1F_1$ are still much shorter than for the inbuilt Matlab function, and the increased time in comparison with

the finite difference method is compensated for by the excellent error properties. The fact that for $n = 20$ the ℓ_∞ error touches the order of 10^{-12} illustrates this. However, the major drawback with this method is that increasing the number of mesh points does not decrease the size of the errors, as was the case with the finite difference and shooting methods. Therefore, this method is very useful for a small number of mesh points, although for large values of n the shooting method would be expected to perform better.

Below is a table showing the corresponding results for ${}_2F_1(-0.9, 0.3; -0.2; x)$, with the errors on $[-1, 1]$ and $[-1, 0]$ shown:

n	Time taken	'hypergeom' time	ℓ_1	ℓ_2	ℓ_∞	ℓ_1 on $[-1,0]$	ℓ_2 on $[-1,0]$	ℓ_∞ on $[-1,0]$
10	0.067913	0.367939	0.3286	0.1362	0.07238	0.07354	0.04582	0.03870
20	0.095184	0.562589	0.1627	0.05900	0.03454	0.008462	0.003639	0.002260
30	0.140470	0.898524	0.4852	0.1079	0.02877	0.1075	0.03733	0.01997
50	0.155365	1.673386	0.5673	0.09628	0.01962	0.1267	0.03384	0.01427
100	0.215859	2.901258	0.1912	0.03217	0.009801	0.01309	0.002459	7.459e-4
200	0.237190	4.421957	0.2208	0.02490	0.005665	0.03000	0.003976	8.561e-4

As in the two previous methods, the fact that two singular points are in the region of integration reduces the effectiveness of the Chebyshev differentiation matrix method for computing ${}_2F_1$. However, it is not as affected by the presence of the singular points, and we find that the ℓ_2 error decreases for large n as n increases, despite the increased length of the vector. But although the errors in the interval $[0, 1]$ seem to be smoothed out, a large number of mesh points may well need to be taken to obtain a workable approximation.

6 ℓ_2 approximation

6.1 Principle of ℓ_2 approximation

The idea of this method is to approximate a function, $f(x)$, by a series up to n powers of x. We do this by defining an **inner product**, creating a set of **orthogonal polynomials** $\{\phi_i : i = 0, 1, ..., n\}$ in that inner product, and then establishing coefficients $\{a_i : i = 0, 1, ..., n\}$ such that:

$$\sum_{i=0}^{n} a_i \phi_i$$

gives the best ℓ_2 approximation to the function in the set spanned by polynomials of degree n.

The first task here is to define an inner product, $< \cdot, \cdot >$, thus:

$$< u, v >= \int_a^b w(x)u(x)v(x)\mathrm{d}x$$

where the region in which we wish to approximate $f(x)$ is $[a, b]$, and $w(x)$ is a certain **weight function**, which is greater than or equal to zero at all values of x.

One possible choice of the weight function is $w(x) = \frac{1}{\sqrt{1-x^2}}$, which would lead to the Chebyshev polynomials $T_i(x)$. However in this case, the simpler choice $w(x) = 1$ will be made. This will result in the orthogonal polynomials obtained being the **Legendre polynomials**, with the coefficient of the highest power of x scaled to 1.

We now wish to choose $\phi_i \in \Pi_i$ for $i = 0, 1, ..., n$, where Π_i denotes the space of polynomials of degree i that are orthogonal to each other. From this choice of inner

product, we may now find the orthogonal polynomials. It can be shown by the **Gram-Schmidt process** that:

$$\phi_i = x^i - \sum_{j=0}^{i-1} \frac{< x^i, \phi_j >}{< \phi_j, \phi_j >} \phi_j \ . \tag{6.1}$$

So if ϕ_0 is chosen to be 1, then:

$$\phi_0 = 1,\ \phi_1 = x,\ \phi_2 = x^2 - \frac{1}{3},\ \phi_3 = x^3 - \frac{3}{5}x,\ \phi_4 = x^4 - \frac{6}{7}x^2 + \frac{3}{35} \ .$$

It can be shown that for $k \geq 1$, the following recurrence relation holds:

$$\phi_{k+1} = x - \frac{< \phi_k, x\phi_k >}{< \phi_k, \phi_k >} \phi_k - \frac{< \phi_k, \phi_k >}{< \phi_{k-1}, \phi_{k-1} >} \phi_{k-1} \ . \tag{6.2}$$

From this, we now need to determine the coefficients a_i. We first define the set $A = span\{\phi_0, \phi_1, ..., \phi_n\}$, and write the ℓ_2 polynomial approximation as:

$$p = \sum_{j=0}^{n} a_j \phi_j \ . \tag{6.3}$$

It can be shown that p is the best ℓ_2 approximation to the function $f(x)$ iff:

$$< f - p, q >= 0,\ \forall q \in A \ .$$

So, letting $q = \phi_i,\ i = 0, 1, ..., n$ in turn, and substituting in the series expression for p in Equation (6.3), and using basic properties of inner products, we arrive at:

$$\sum_{j=0}^{n} < \phi_i, \phi_j > a_j =< f, \phi_i >,\ i = 0, 1, ..., n$$

and, as the polynomials ϕ_i are orthogonal and so $< \phi_i, \phi_j >= 0,\ \forall i \neq j$, we have:

$$a_j = \frac{< f, \phi_j >}{< \phi_j, \phi_j >} \ . \tag{6.4}$$

So the expansion in terms of the first n orthogonal polynomials is:

$$f(x) \approx p(x) = \sum_{j=0}^{n} \frac{< f, \phi_j >}{< \phi_j, \phi_j >} \phi_j \ . \tag{6.5}$$

The Matlab code used to find the ℓ_2 approximation is straightforward. The Matlab function '**quad**' is used to apply quadrature to find the integrals $< f, \phi_j >$ and $< \phi_j, \phi_j >$ for $j = 0, 1, ..., n$ for appropriate n that we choose. Then the terms are summed to find the polynomial approximation p, at which point this may be compared with the solution generated by the inbuilt Matlab function for computing the desired functions.

When analysing the Matlab results, two important points must be noted:

- Establishing the ℓ_2 approximation of the function $f(x)$ involves computing integrals in terms of f itself multiplied by the orthogonal polynomials. As the reason we are computing the hypergeometric functions ${}_1F_1$ and ${}_2F_1$ using the methods in this project is that they take a long time to be computed by the Matlab function, 'hypergeom', and hence the integrals will take a long time to be computed using 'quad', so this method is only viable if these integrals have values which have been previously computed and stored. Otherwise, computing the approximation will take longer than evaluating the hypergeometric functions themselves using Matlab.
- The polynomial approximation obtained using this method in terms of the first n orthogonal polynomials will in general not be the same as the truncated Taylor series expansion of n terms. Therefore, in the analysis, it is worthwhile to compare the accuracy of the ℓ_2 approximation and the truncated Taylor series expansion.

The code used is shown in Appendix D.

6.2 Numerical results

The fourth order polynomials generated by ℓ_2 approximation are shown for each of the three functions:

Function being approximated	ℓ_2 expansion and Taylor series (in bold), up to x^4 (coefficients to 6s.f.)
$J_0(x)$	$0.999991 - 0.249805x^2 + 0.0150405x^4$ $\mathbf{1 - 0.25x^2 + 0.015625x^4}$
${}_1F_1(0.1; 0.2; x)$	$1.00001 + 0.499155x + 0.228904x^2 + 0.0768327x^3 + 0.0184428x^4$ $\mathbf{1 + 0.5x + 0.229167x^2 + 0.0729167x^3 + 0.0176595x^4}$
${}_2F_1(-0.9, 0.3; -0.2; x)$	$1.00327 + 1.32523x + 0.0555430x^2 + 0.132313x^3 + 0.138537x^4$ $\mathbf{1 + 1.35x + 0.109688x^2 + 0.0513906x^3 + 0.0317979x^4}$

Below is a table showing the ℓ_1, ℓ_2 and ℓ_∞ errors for each of the three functions on $[-1, 1]$ for 50, 100 and 200 mesh points, as well as a comparison of the computation times with the inbuilt Matlab function. The errors for the computation of ${}_2F_1$ are also given on the interval $[-1, 0.6]$):

Function	n	Time taken	Matlab time	ℓ_1	ℓ_2	ℓ_∞
$J_0(x)$	50	0.095184	0.000632	6.298	1.038	2.849e-5
$J_0(x)$	100	0.140470	0.000925	12.39	1.440	2.849e-5
$J_0(x)$	200	0.155365	0.001455	24.60	2.017	2.849e-5
${}_1F_1(0.1; 0.2; x)$	50	1.817793	0.896300	28.10	4.686	4.978e-4
${}_1F_1(0.1; 0.2; x)$	100	2.399744	1.637009	55.59	6.547	4.978e-4
${}_1F_1(0.1; 0.2; x)$	200	2.652233	2.813293	110.6	9.202	4.978e-4
${}_2F_1(-0.9, 0.3; -0.2; x)$	50	7.020790	1.673386	71.38/0.1754	11.79/0.03171	0.1512/0.01256
${}_2F_1(-0.9, 0.3; -0.2; x)$	100	7.108385	2.901258	141.1/0.3458	16.47/0.04391	0.1512/0.01256
${}_2F_1(-0.9, 0.3; -0.2; x)$	200	7.194987	4.421957	280.5/0.6876	23.14/0.06150	0.1512/0.01256

The fact that the function itself is required to compute the ℓ_2 approximation to it, as explained earlier, means that the time taken to calculate the approximation is actually greater than that taken by the inbuilt Matlab function in all three cases. The only two cases where this method would be applicable therefore are when the required integrals are known or when a value for the function is required for a large number of points, so that in each case the ℓ_2 approximation may be used instead. However, even if this were the case, the approximation for each of the functions is far worse than for any of the three differential equation methods used previously. The weakness of the differential equation methods used was generally their accuracy near the boundary $x = 1$, though as this approximation is in the form of a series in x, the greatest error would be likely to occur when the higher order terms ignored contribute the most, i.e. near $x = \pm 1$, so this method does not improve on this drawback.

Below is a graph showing results for ${}_2F_1$. The error graph also contains the error of the Taylor series expansion up to the x^4 term.

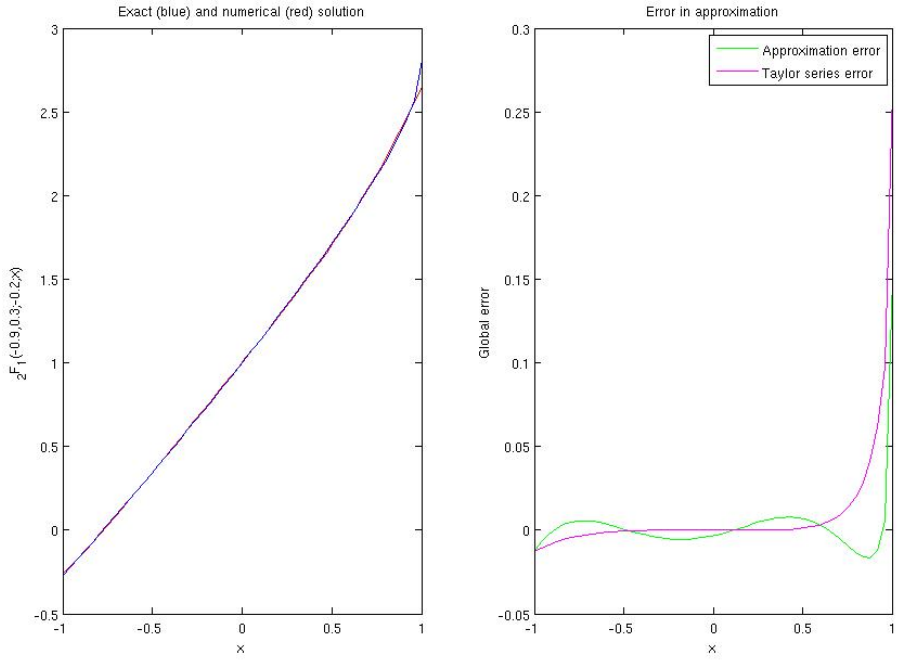

Figure 6: Numerical results for ℓ_2 approximation applied to ${}_2F_1$ with 50 mesh points

Comparing the error in the ℓ_2 approximation of ${}_2F_1$ with the error in the first 5 terms of the Taylor series expansion, it can be seen that in the middle of the interval $[-1, 1]$, the Taylor series expansion is more accurate, but the error near the end-points is substantially larger than the ℓ_2 approximation error. Similar results are observed when approximating $J_0(x)$ and ${}_1F_1$.

7 Padé approximants

7.1 Construction of Padé approximants

The principle of constructing a **Padé approximant**, a form of **rational approximation**, on the interval $[-1, 1]$ to a function $f(x)$ with Taylor series expansion:

$$f(x) = \sum_{j=1}^{\infty} A_j x^j$$

is to approximate f by a ratio of polynomials with small degrees such that the ratio approximates f to as high a number of powers of x as possible.

Let $R_{m,n}(a, b)$ be the space of rational polynomials with $r(x) = \frac{q(x)}{d(x)}$, with $q(x) \in \Pi_m$, $d(x) \in \Pi_n$, where Π_k denotes the set of polynomials of degree less than or equal to k. If

we write:

$$q(x) = q_1 + q_2x + q_3x^2 + ... + q_{m+1}x^m = \sum_{j=1}^{m+1} q_j x^{j-1} \tag{7.1}$$

$$d(x) = 1 + d_2x + d_3x^2 + ... + d_{n+1}x^n = 1 + \sum_{j=2}^{n+1} d_j x^{j-1} \tag{7.2}$$

then the goal is to find coefficients q_1,q_2,...,q_{m+1},d_1,d_2,...,d_{m+1} such that:

$$u - \frac{q}{d} = \sum_{j=s}^{\infty} C_j x^j$$

for some coefficients C_j, where s is as large as possible.

This method should work well in the interval $[-1, 1]$ which we will be using to approximate all three functions, as the coefficients of x^j in the above expression will decrease as j increases. We would expect the approximation to work the best when $|x|$ is small, i.e. in the middle of the interval, as this will result in $\sum_{j=s}^{\infty} C_j x^j$ being small.

To construct the Padé approximant, we first return to the Taylor expansion for f:

$$f(x) = A_1 + A_2x + A_3x^2 + ... = \sum_{j=1}^{\infty} A_j x^{j-1} \ .$$

If we first try to approximate f by a function in $R_{2,1}[-1, 1]$:

$$f(x) \approx \frac{q(x)}{d(x)} = \frac{q_1 + q_2x + q_3x^2}{1 + d_2x}$$

then maximising the order of $f - \frac{q}{d}$ is equivalent to doing the same for $q - df$, where:

$$q(x) - d(x)f(x) = (q_1 + q_2x + q_3x^2) - (1 + d_2x)(A_1 + A_2x + A_3x^2 + ...) \ . \tag{7.3}$$

Eliminating terms of order 1, x, x^2 and x^3 from this expansion gives four equations which can be solved by direct substitution to find:

$$q_1 = A_1, \ q_2 = -\frac{A_1A_4}{A_3} + A_2, \ q_3 = -\frac{A_2A_4}{A_3} + A_3, \ d_2 = -\frac{A_4}{A_3} \ . \tag{7.4}$$

Alternatively, finding an approximation in $R_{3,2}$ gives us that:

$$q(x) - d(x)f(x) = (q_1 + q_2x + q_3x^2 + q_4x^3) - (1 + d_2x + d_3x^2)(A_1 + A_2x + A_3x^2 + ...)$$

which yields six equations which can be written as the matrix system:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & -A_1 & 0 \\ 0 & 0 & 1 & 0 & -A_2 & -A_1 \\ 0 & 0 & 0 & 1 & -A_3 & -A_2 \\ 0 & 0 & 0 & 0 & -A_4 & -A_3 \\ 0 & 0 & 0 & 0 & -A_5 & -A_4 \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \\ A_5 \\ A_6 \end{bmatrix}$$

and similarly a 7×7 system of equations can be obtained for an approximation in $R_{4,2}$. Solving these systems will give us the Padé approximants, from which we can examine errors in the approximations.

It should be noted that the function we test on, the Bessel function, $J_0(x)$, is even, so the approximation to it will not have terms in odd powers of x. We therefore construct approximants belonging to $R_{4,2}$ and $R_{6,2}$ of the form:

$$f(x) \approx \frac{q_1 + q_3x^2 + q_5x^4}{1 + d_3x^2},\ f(x) \approx \frac{q_1 + q_3x^2 + q_5x^4 + q_7x^6}{1 + d_3x^2}$$

to arrive at and solve 4×4 and 5×5 matrix systems respectively.

The code used to compute and plot these approximations was simple. First, the required terms in the Taylor series for the function to be approximated were computed, and the mesh grid we wished to solve on was defined. Then, either the required coefficients for the polynomials in the Padé approximants were directly computed, or a matrix system for them was set up and Matlab asked to solve the system. Having done this, Matlab was asked to compute the approximant at each value of x on the mesh grid defined, and the results could then be compared with the true solution of the function we are trying to evaluate.

The full code used to generate and solve matrix systems for each of the Padé approximants is in Appendix E.

7.2 Numerical results

Below are the Padé approximants in various spaces for the three functions being approximated:

Function	Approximation space	Expansion (coefficients to 6s.f.)
$J_0(x)$	$R_{4,2}$	$\frac{1-0.222222x^2+0.00868056x^4}{1+0.0278889x^2}$
$J_0(x)$	$R_{6,2}$	$\frac{1-0.234375x^2+0.117188x^4-0.000189887x^6}{1+0.0156250x^2}$
${}_1F_1(0.1;0.2;x)$	$R_{2,1}$	$\frac{1+0.1818x+0.0700758x^2}{1-0.318182x}$
${}_1F_1(0.1;0.2;x)$	$R_{3,2}$	$\frac{1+0.108189x+0.0808686x^2+0.00693036x^3}{1-0.391811x+0.0476076x^2}$
${}_1F_1(0.1;0.2;x)$	$R_{4,2}$	$\frac{1+0.172620x+0.0974797x^2+0.0138936x^3+0.00112209x^4}{1-0.3327380x+0.0320032x^2}$
${}_2F_1(-0.9,0.3;-0.2;x)$	$R_{2,1}$	$\frac{1+0.881481x-0.522813x^2}{1-0.468519x}$
${}_2F_1(-0.9,0.3;-0.2;x)$	$R_{3,2}$	$\frac{1+0.390107x-1.02637x^2+0.161875x^3}{1-0.959893x+0.159832x^2}$
${}_2F_1(-0.9,0.3;-0.2;x)$	$R_{4,2}$	$\frac{1+0.200051x-1.16531x^2+0.299785x^3+0.00313185x^4}{1-1.14995x+0.277429x^2}$

Below is a table showing the numerical results for the approximation of the Bessel function $J_0(x)$ on $[-1,1]$:

Approximation space	n	Time taken	'besselj' time	ℓ_1	ℓ_2	ℓ_∞
$R_{4,2}$	20	0.000628	0.000460	1.639e-5	7.766e-6	5.016e-6
$R_{6,2}$	20	0.000807	0.000460	1.075e-7	5.533e-8	3.699e-8
$R_{4,2}$	50	0.000638	0.000632	3.285e-5	1.005e-5	5.016e-6
$R_{6,2}$	50	0.000845	0.000632	2.050e-7	6.899e-8	3.699e-8
$R_{4,2}$	100	0.000767	0.000925	6.083e-5	1.321e-5	5.016e-6
$R_{6,2}$	100	0.000917	0.000925	3.727e-7	8.912e-8	3.699e-8
$R_{4,2}$	200	0.000762	0.001455	1.170e-4	1.799e-5	5.016e-6
$R_{6,2}$	200	0.000945	0.001455	7.099e-7	1.202e-7	3.699e-8

The approximation here, especially in the $R_{6,2}$ space, is fairly accurate and, for large values of n, done in a short period of time in comparison with the inbuilt Matlab solver, 'besselj'. However, the approximation gained by solving the differential equations using the shooting method or Chebyshev differentiation method (although not the finite difference method) is far more accurate.

However, all the differential equation methods are more accurate (but are also slower) when approximating the ${}_1F_1$ function. The results for this using Padé approximants are shown below:

Approximation space	n	Time taken	'hypergeom' time	ℓ_1	ℓ_2	ℓ_∞
$R_{2,1}$	20	0.000883	0.104515	0.03462	0.01541	0.01227
$R_{3,2}$	20	0.000892	0.104515	2.748e-4	1.448e-4	1.261e-4
$R_{4,2}$	20	0.000931	0.104515	2.222e-5	1.193e-5	1.051e-5
$R_{2,1}$	50	0.000951	0.896300	0.07598	0.02141	0.01227
$R_{3,2}$	50	0.001090	0.896300	5.705e-4	1.920e-4	1.261e-4
$R_{4,2}$	50	0.001109	0.896300	4.509e-5	1.553e-5	1.051e-5
$R_{2,1}$	100	0.001141	1.637009	0.1454	0.02893	0.01227
$R_{3,2}$	100	0.001212	1.637009	0.001070	2.547e-4	1.261e-4
$R_{4,2}$	100	0.001237	1.637009	8.391e-7	2.045e-5	1.051e-5
$R_{2,1}$	200	0.001222	2.813293	0.2845	0.03997	0.01227
$R_{3,2}$	200	0.001235	2.813293	0.002073	3.485e-4	1.261e-4
$R_{4,2}$	200	0.001246	2.813293	1.162e-4	2.787e-5	1.051e-5

The graph below shows the results for computing ${}_1F_1$:

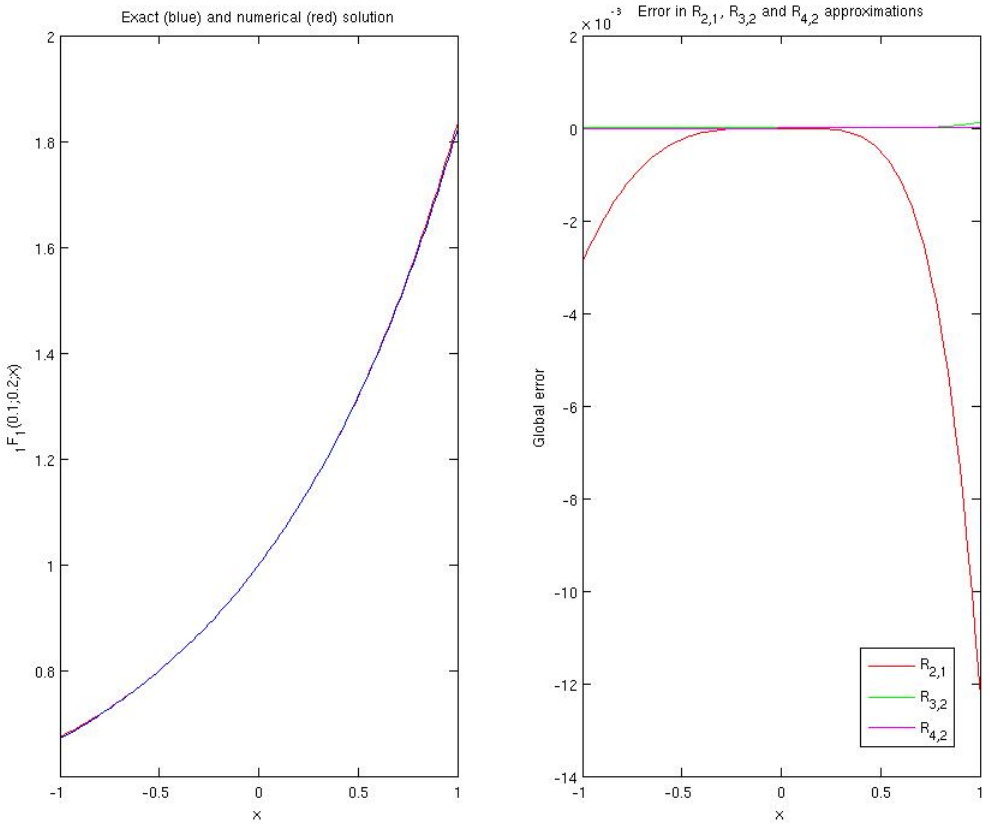

Figure 7: Numerical results for Padé approximants applied to ${}_1F_1$ with 100 mesh points

The results for ${}_2F_1$, including the errors on the interval $[-1, 0.6]$ as well as those on the interval $[-1, 1]$, are shown:

Space $R_{m,n}$	n	Time taken	'hypergeom' time	ℓ_1	ℓ_2	ℓ_∞	ℓ_1 on $[-1,0]$	ℓ_2 on $[-1,0]$	ℓ_∞ on $[-1,0]$
$R_{2,1}$	20	0.000867	0.562589	0.3257	0.2539	0.2497	0.01800	0.004381	0.002559
$R_{3,2}$	20	0.000947	0.562589	0.1910	0.1774	0.1771	3.188e-4	1.539e-4	1.171e-4
$R_{4,2}$	20	0.000962	0.562589	0.1645	0.1575	0.1574	6.987e-5	3.595e-5	2.852e-5
$R_{2,1}$	50	0.001163	1.673386	0.5238	0.2747	0.2497	0.02426	0.006298	0.002828
$R_{3,2}$	50	0.001215	1.673386	0.2471	0.1815	0.1771	6.825e-4	2.158e-4	1.414e-4
$R_{4,2}$	50	0.001262	1.673386	0.1992	0.1596	0.1574	1.465e-4	5.023e-5	3.555e-5
$R_{2,1}$	100	0.001672	2.901258	0.8736	0.3139	0.2497	0.04673	0.008589	0.002938
$R_{3,2}$	100	0.001718	2.901258	0.3558	0.1927	0.1771	0.001288	2.884e-4	1.501e-4
$R_{4,2}$	100	0.001739	2.901258	0.2706	0.1662	0.1574	2.739e-4	6.659e-5	3.813e-5
$R_{2,1}$	200	0.001789	4.421257	1.5860	0.3850	0.2497	0.09169	0.01192	0.002993
$R_{3,2}$	200	0.001822	4.421257	0.5845	0.2176	0.1771	0.002500	3.953e-4	1.546e-4
$R_{4,2}$	200	0.001879	4.421257	0.4237	0.1827	0.1574	5.287e-4	9.076e-5	3.946e-5

Again, when approximating ${}_2F_1$ the approximation near $x = 1$ has large errors, which is to be expected as the method is based on removing higher order terms in the Taylor series of the function; these terms would be expected to contribute more near the end-points of the interval than near the centre. The errors in the left 80% of the interval show that a decent approximation is given away from $x = 1$, although this is still not as effective as the finite difference and shooting methods.

8 Conclusion

The results of the investigation into the five numerical methods tested show that the methods used which involved solving the ordinary differential equations satisfied by the hypergeometric functions are generally more effective than those which used the Taylor series to evaluate the approximation. The three errors measured were usually smaller for the differential equation methods than for the series methods, especially the ℓ_1 and ℓ_2 errors. Also, apart from the Chebyshev differentiation matrix approach, the differential equations could be solved on any interval chosen as opposed to just $[-1, 1]$, although if this were tried on ${}_2F_1$ outside the interval, the function would not converge there. The two series methods would only have given good approximations within the interval $[-1, 1]$, as otherwise the terms neglected would come into play. Further, as opposed to ℓ_2 approximation, none of the differential equation methods use properties of the function apart from boundary conditions, meaning that useful approximations could be gained with the differential equation methods even if the user of the program knew little about the problem.

The fact that for ℓ_2 approximation, prior information was needed about the solution, resulted in a large computation time, and so this form of approximation would only be useful if values at a large number of points were needed and an initial computation of the function had already been stored, so a number of simple computations could follow. However, as well as this drawback, the ℓ_2 approximation method seemed to have the least desirable error properties out of the five, so high order polynomials would need to be computed for it to compete with the other methods. Using Padé approximants fared better, especially for approximating the Bessel function, where the accuracy was impressive, and took less time, though the polynomials on numerator and denominator would also need to be of relatively high order for this approach to be preferable to the differential equation methods for computing ${}_1F_1$ and ${}_2F_1$.

Of the three differential equation methods, the advantages of using finite differences or the shooting method with RK4 include the facts that the differential equation can be solved on any interval, if appropriate boundary conditions are known, rather than just $[-1, 1]$ as in the case of Chebyshev differentiation matrices, and also that the error is predicted to decrease as the number of mesh points increases. Further, the time taken to compute the approximation is substantially less with these two methods, especially with finite differences, whereas the shooting method error decreases more rapidly as the number of mesh points increases and the errors for a very large number of mesh points are excellent. However the computed solutions are less accurate than for the Chebyshev

differentiation matrix method for a small number of mesh points, especially for the ${}_2F_1$ function close to $x = 1$. The Chebyshev differentiation matrix method takes longer to arrive at the solutions, due to the fact that a dense matrix system needs to be computed, but the accuracy is much improved for a small number of mesh points as well as for ${}_2F_1$. Hence, for small numbers of mesh points, the Chebyshev differentiation matrix method seems preferable, but the shooting method is most likely to perform better for a large number of mesh points. However a new method, possibly combining the two could be devised, and no individual method seems to work well for ${}_2F_1$ on $[0, 1]$, due to the two singular points in that interval.

A Appendix A: Matlab code for using the finite difference method to approximate the hypergeometric function ${}_1F_1$

This code illustrates the method for finding ${}_1F_1(0.1; 0.2; x)$ on $[-1, 1]$ with 200 mesh points via solving the ordinary differential equation $xy'' + (c - x)y' - ay = 0$ with appropriate boundary conditions; the code is similar for finding the Bessel function $J_0(x)$ and ${}_2F_1(-0.9, 0.3; -0.2, x)$, and for calculating the functions with different numbers of mesh points.

```
% Define interval, number of mesh points and step size
    n = 200;
    x = linspace(-1,1,n)';
    h = x(2)-x(1);

% Define a,c
    a=0.1;
    c=0.2;

tic

% Construct differentiation matrix
    v1 = ones(n,1);
    D2 = (diag(-2*v1) + diag(v1(2:end),1)+diag(v1(2:end),-1))/h^2;
    D1 = (-diag(v1(1:end-1),-1) + diag(v1(2:end),1))/(2*h);

% Multiply differentiation matrices by appropriate factors
    B=zeros(n,n);
    for j=1:n
        B(:,j)=x;
    end

% Differential operator for finite differences
    Mfd = B.*D2+c*D1-B.*D1-a*eye(n);
    Mfd(1,:) = [1 zeros(1,n-1)];
    Mfd(n,:) = [zeros(1,n-1) 1];

% Calculate right-hand side of system of equations
    d = zeros(n,1);
    d(1)=0.670954857323271; % BC at x=-1
    d(end)=1.823844396378180; % BC at x=1

% Find numerical solution
    yfd = Mfd\d; toc

% Find errors in various norms
    e1 = norm(hypergeom(a,c,x)-yfd,1)
    e2 = norm(hypergeom(a,c,x)-yfd,2)
    einf=norm(hypergeom(a,c,x)-yfd,inf)
```

B Appendix B: Matlab code for using the shooting method to approximate the hypergeometric function ${}_1F_1$

This code illustrates the method for finding ${}_1F_1(0.1; 0.2; x)$ on $[-1, 1]$ with 201 mesh points via solving the ordinary differential equation $xy'' + (c - x)y' - ay = 0$ with appropriate boundary conditions; the code is similar for finding the Bessel function $J_0(x)$, and for calculating the functions with different numbers of mesh points, code for ${}_2F_1$ different.

```
% Solve equation on [-1,0]
    a=0.1;
    c=0.2;

% Set up equation
    A = @(x,y) x ;
    B = @(x,y) c-x ;
    C = @(x,y) -a ;
    f1 = @(x,y,z) z ;
    f2 = @(x,y,z) -1/A(x,y)*(B(x,y)*z+C(x,y)*y) ;

% Boundary conditions
    alpha=0.670954857323271; % BC at x=-1
    beta=1; % BC at x=0
    x0=-1 ;
    y0=alpha ;
    xf=0 ;
    yf=beta ;

% Guess of initial derivatives for shooting method
    dydx1 = 0 ;
    dydx2 = 1 ;

% Number of mesh points and step-size
    n=101 ;
    h=(xf-x0)/n ;
    x=zeros(n,1);
    for i = 1:n+1
        x(i)=x0+(i-1)*h;
    end

% Set up ICs for y1 and y1'
    y1(1)=y0;
    z1(1)=dydx1;

    tic
% Using 4th order Runge-Kutta method
    for i = 1:n
        rk1y=f1(x(i),y1(i),z1(i));
        rk1z=f2(x(i),y1(i),z1(i));
        rk2y=f1(x(i)+0.5*h,y1(i)+0.5*rk1y*h,z1(i)+0.5*rk1z*h);
        rk2z=f2(x(i)+0.5*h,y1(i)+0.5*rk1y*h,z1(i)+0.5*rk1z*h);
        rk3y=f1(x(i)+0.5*h,y1(i)+0.5*rk2y*h,z1(i)+0.5*rk2z*h);
        rk3z=f2(x(i)+0.5*h,y1(i)+0.5*rk2y*h,z1(i)+0.5*rk2z*h);
        rk4y=f1(x(i)+h,y1(i)+rk3y*h,z1(i)+rk3z*h);
```

```
        rk4z=f2(x(i)+h,y1(i)+rk3y*h,z1(i)+rk3z*h);
        y1(i+1)=y1(i)+h/6*(rk1y+2*rk2y+2*rk3y+rk4y);
        z1(i+1)=z1(i)+h/6*(rk1z+2*rk2z+2*rk3z+rk4z);
    end

% ICs for y2 and y2', and RK4 method
    y2(1)=0;
    z2(1)=dydx2;
    for i = 1:n
        rk1y=f1(x(i),y2(i),z2(i));
        rk1z=f2(x(i),y2(i),z2(i));
        rk2y=f1(x(i)+0.5*h,y2(i)+0.5*rk1y*h,z2(i)+0.5*rk1z*h);
        rk2z=f2(x(i)+0.5*h,y2(i)+0.5*rk1y*h,z2(i)+0.5*rk1z*h);
        rk3y=f1(x(i)+0.5*h,y2(i)+0.5*rk2y*h,z2(i)+0.5*rk2z*h);
        rk3z=f2(x(i)+0.5*h,y2(i)+0.5*rk2y*h,z2(i)+0.5*rk2z*h);
        rk4y=f1(x(i)+h,y2(i)+rk3y*h,z2(i)+rk3z*h);
        rk4z=f2(x(i)+h,y2(i)+rk3y*h,z2(i)+rk3z*h);
        y2(i+1)=y2(i)+h/6*(rk1y+2*rk2y+2*rk3y+rk4y);
        z2(i+1)=z2(i)+h/6*(rk1z+2*rk2z+2*rk3z+rk4z);
    end

% Find numerical solution on [-1,0]
    y3=y1+((beta-y1(end))/y2(end))*y2; toc

%%
% Find numerical solution on [0,1] by working backwards from x=1 to x=0
    a=0.1;
    c=0.2;

% Boundary conditions
    alpha=1.823844396378180; % BC at x=1
    beta=1; % BC at x=0
    x0=0 ;
    y0=alpha ;
    xf=1 ;
    yf=beta ;

% Set up equation using substitution of X=1-x to avoid singularity at x=0
    A = @(x,y) (1-x) ;
    B = @(x,y) -(c-(1-x)) ;
    C = @(x,y) -a ;
    f1 = @(x,y,z) z ;
    f2 = @(x,y,z) -1/A(x,y)*(B(x,y)*z+C(x,y)*y) ;

% Guess of derivatives
    dydx1 = 0 ;
    dydx2 = 1 ;

% Number of steps and step-size
    n=101 ;
    h=(xf-x0)/n ;
    xnew=zeros(n,1);
    X=zeros(n,1);
    for i = 1:n+1
        xnew(i)=x0+(i-1)*h;
```

```
        X(i)=xf-xnew(i);
    end

% Set initial conditions and apply RK4 method
    y1(1)=y0;
    z1(1)=dydx1;

    tic

    for i = 1:n
        rk1y=f1(xnew(i),y1(i),z1(i));
        rk1z=f2(xnew(i),y1(i),z1(i));
        rk2y=f1(xnew(i)+0.5*h,y1(i)+0.5*rk1y*h,z1(i)+0.5*rk1z*h);
        rk2z=f2(xnew(i)+0.5*h,y1(i)+0.5*rk1y*h,z1(i)+0.5*rk1z*h);
        rk3y=f1(xnew(i)+0.5*h,y1(i)+0.5*rk2y*h,z1(i)+0.5*rk2z*h);
        rk3z=f2(xnew(i)+0.5*h,y1(i)+0.5*rk2y*h,z1(i)+0.5*rk2z*h);
        rk4y=f1(xnew(i)+h,y1(i)+rk3y*h,z1(i)+rk3z*h);
        rk4z=f2(xnew(i)+h,y1(i)+rk3y*h,z1(i)+rk3z*h);
        y1(i+1)=y1(i)+h/6*(rk1y+2*rk2y+2*rk3y+rk4y);
        z1(i+1)=z1(i)+h/6*(rk1z+2*rk2z+2*rk3z+rk4z);
    end

    y2(1)=0;
    z2(1)=dydx2;
    for i = 1:n
        rk1y=f1(xnew(i),y2(i),z2(i));
        rk1z=f2(xnew(i),y2(i),z2(i));
        rk2y=f1(xnew(i)+0.5*h,y2(i)+0.5*rk1y*h,z2(i)+0.5*rk1z*h);
        rk2z=f2(xnew(i)+0.5*h,y2(i)+0.5*rk1y*h,z2(i)+0.5*rk1z*h);
        rk3y=f1(xnew(i)+0.5*h,y2(i)+0.5*rk2y*h,z2(i)+0.5*rk2z*h);
        rk3z=f2(xnew(i)+0.5*h,y2(i)+0.5*rk2y*h,z2(i)+0.5*rk2z*h);
        rk4y=f1(xnew(i)+h,y2(i)+rk3y*h,z2(i)+rk3z*h);
        rk4z=f2(xnew(i)+h,y2(i)+rk3y*h,z2(i)+rk3z*h);
        y2(i+1)=y2(i)+h/6*(rk1y+2*rk2y+2*rk3y+rk4y);
        z2(i+1)=z2(i)+h/6*(rk1z+2*rk2z+2*rk3z+rk4z);
    end

% Compute approximate solution on [0,1]
    y3new=y1+((beta-y1(end))/y2(end))*y2; toc

% Reverse order of vector used
    x1=X(1:end-1);
    y31=y3new(1:end-1);

%% Combine solutions for [-1,0] and [0,1]
    for l=1:length(x1)
        x1new(l)=x1(length(x1)-l+1);
        y31new(l)=y31(length(x1)-l+1);
    end

%% Joining up of vectors and plots
    x2=[x,x1new];
    y32=[y3,y31new];
    plot(x2,y32); hold on
    plot(x2,hypergeom(a,c,x2)); hold off
```

C Appendix C: Matlab code for using Chebyshev differentiation matrices to approximate the hypergeometric function ${}_2F_1$

This code illustrates the method for finding ${}_2F_1(-0.9, 0.3; -0.2; x)$ on $[-1, 1]$ with 200 mesh points via solving the ordinary differential equation $x(1-x)y'' + [c - (a+b+1)x]y' - aby = 0$ with appropriate boundary conditions; the code is similar for finding the Bessel function $J_0(x)$ and ${}_1F_1(0.1, 0.2; x)$, and for calculating the functions with different numbers of mesh points.

```
% Number of mesh points
    n=200;

% Vector of Chebyshev points
    x = -cos((0:n-1)*pi/(n-1))';

% Set up matrices labelled (5.1) and (5.3) in text
    A = zeros(n,n);
    B = zeros(n,n);
    tic
    for k = 0:n-1
        A(:,k+1) = cos(acos(x)*k); % Chebyshev polys
        B(:,k+1) = k*sin(acos(x)*k)./sqrt(1-x.^2); % Derivative of Ch'v polys
        B(1,k+1) = (-1)^(k+1)*k^2;
        B(end,k+1) = k^2;
    end

% Find differentiation matrix
    D = B/A;

% Find appropriate matrices corresponding to coefficients of derivatives in ODE
    X=x.*(1-x);
    E=zeros(n,n);
    for j=1:n
        E(:,j)=x;
    end
    F=zeros(n,n);
    for k=1:n
        F(:,k)=X;
    end

% Set up values for a,b,c
    a=-0.9;
    b=0.3;
    c=1+a-b;

% Set up matrix corresponding to left-hand side of ODE
    Mch=F.*D^2+c*D-(a+b+1)*E.*D-a*b*eye(n);
    Mch(1,:) = [1 zeros(1,n-1)];
    Mch(n,:) = [zeros(1,n-1) 1];

% Right-hand side vector with boundary conditions
    d = zeros(n,1);
    d(1)=hypergeom([a,b],c,-1);
```

```
    d(end)=hypergeom([a,b],c,1);

% Numerical solution
    u=Mch\d; toc

% Compute various errors
    e1 = norm(hypergeom([a,b],c,x)-u,1)
    e2 = norm(hypergeom([a,b],c,x)-u,2)
    einf=norm(hypergeom([a,b],c,x)-u,inf)
```

D Appendix D: Matlab code for using ℓ_2 approximation to approximate the hypergeometric function ${}_2F_1$

This code illustrates the method for approximating ${}_2F_1(-0.9, 0.3; -0.2; x)$ on $[-1, 1]$ by finding the first 5 terms in a series expansion of orthogonal polynomials, in such a way that the expansion minimises the ℓ_2 error over linear combinations of these 5 polynomials. Finding the ℓ_2 approximation for Bessel and ${}_1F_1$ functions can be done similarly.

```
% Define a,b,c and mesh points
    a=-0.9; b=0.3; c=1+a-b;
    n=50;
    x=linspace(-1,1,n); tic

% Compute relevant integrals
    F1 = @(x)hypergeom([a,b],c,x);Q1=quad(F1,-1,1);
    F2 = @(x)1+0*x; Q2=quad(F2,-1,1);
    F3 = @(x)x.*hypergeom([a,b],c,x);Q3=quad(F3,-1,1);
    F4 = @(x)x.*x;Q4=quad(F4,-1,1);
    F5 = @(x)(x.^2-1/3).*hypergeom([a,b],c,x);Q5=quad(F5,-1,1);
    F6 = @(x)(x.^2-1/3).*(x.^2-1/3);Q6=quad(F6,-1,1);
    F7 = @(x)(x.^3-3/5*x).*hypergeom([a,b],c,x);Q7=quad(F7,-1,1);
    F8 = @(x)(x.^3-3/5*x).*(x.^3-3/5*x);Q8=quad(F8,-1,1);
    F9 = @(x)(x.^4-30/35*x.^2+3/35).*hypergeom([a,b],c,x);Q9=quad(F9,-1,1);
    F10 = @(x)(x.^4-30/35*x.^2+3/35).*(x.^4-30/35*x.^2+3/35);Q10=quad(F10,-1,1);

    toc; plot(x,Q1/Q2+Q3/Q4*x+Q5/Q6*(x.^2-1/3)+Q7/Q8*(x.^3-3/5*x)...
                +Q9/Q10*(x.^4-30/35*x.^2+3/35))

% Find coefficients of powers of x in series expansion
    Q1/Q2-Q5/Q6/3+3*Q9/Q10/35 % coefficient of 1
    Q3/Q4-3*Q7/Q8/5 % coefficient of x
    Q5/Q6-30*Q9/Q10/35 % coefficient of x^{2}
    Q7/Q8 % coefficient of x^{3}
    Q9/Q10 % coefficient of x^{4}

% Compute errors in various norms
    e1 = norm(hypergeom([a,b],c,x)-Q1/Q2+Q3/Q4*x+Q5/Q6*(x.^2-1/3)...
                +Q7/Q8*(x.^3-3/5*x)+Q9/Q10*(x.^4-30/35*x.^2+3/35),1)
    e2 = norm(hypergeom([a,b],c,x)-Q1/Q2+Q3/Q4*x+Q5/Q6*(x.^2-1/3)...
                +Q7/Q8*(x.^3-3/5*x)+Q9/Q10*(x.^4-30/35*x.^2+3/35),2)
    einf = norm(hypergeom([a,b],c,x)-Q1/Q2+Q3/Q4*x+Q5/Q6*(x.^2-1/3)...
                +Q7/Q8*(x.^3-3/5*x)+Q9/Q10*(x.^4-30/35*x.^2+3/35),inf)
```

E Appendix E: Matlab code for finding Padé approximants to approximate the hypergeometric function ${}_1F_1$

This code illustrates the method for approximating ${}_1F_1(0.1; 0.2; x)$ on $[-1, 1]$ by finding the Padé approximants in $P_{2,1}$,$P_{3,2}$ and $P_{4,2}$. A similar method can be used for Bessel and ${}_2F_1$ functions, and also for finding Padé approximants with polynomials of higher degree on the numerator and denominator.

```
% Define a,c
    a=0.1;
    c=0.2;

% Number of terms in Taylor series to be computed
    n=6;

    tic

% Compute Taylor series terms
    A=zeros(n+1,1);
    A(1)=1;
    for i=2:n+1
        A(i)=gamma(a+i-1)/gamma(a)/gamma(c+i-1)*gamma(c)/factorial(i-1);
    end

    toc

% Define mesh points
    x=linspace(-1,1,100)';

%% f=quadratic/linear
    tic

% Find terms in Pade approximant
    q11=A(1)
    q12=-A(1)*A(4)/A(3)+A(2)
    q13=-A(2)*A(4)/A(3)+A(3)
    d12=-A(4)/A(3)

% Find Pade approximant using these terms
    f1=@(x) (q11+q12*x+q13*x.^2)./(1+d12*x); toc

% Compute errors in various norms
    e1 = norm(hypergeom(a,c,x)-(q11+q12*x+q13*x.^2)./(1+d12*x),1)
    e2 = norm(hypergeom(a,c,x)-(q11+q12*x+q13*x.^2)./(1+d12*x),2)
    einf=norm(hypergeom(a,c,x)-(q11+q12*x+q13*x.^2)./(1+d12*x),inf)


%% f=cubic/quadratic

% Set up matrix and vectors for linear system of equations
    z=zeros(6,1);
    B=zeros(6,6);
    y=zeros(6,1);
```

```
% Define and solve linear system
    tic, z=[A(1),A(2),A(3),A(4),A(5),A(6)]';
    B=[1,0,0,0,0,0;0,1,0,0,-A(1),0;0,0,1,0,-A(2),-A(1);0,0,0,1,-A(3),-A(2);...
0,0,0,0,-A(4),-A(3);0,0,0,0,-A(5),-A(4)];
    y=B\z;

% Define terms in Pade approximant
    q21=y(1)
    q22=y(2)
    q23=y(3)
    q24=y(4)
    d22=y(5)
    d23=y(6)

% Find Pade approximant using these terms
    f2=@(x) (q21+q22*x+q23*x.^2+q24*x.^3)./(1+d22*x+d23*x.^2); toc

% Find various errors
    e1 = norm(hypergeom(a,c,x)...
                -(q21+q22*x+q23*x.^2+q24*x.^3)./(1+d22*x+d23*x.^2),1)
    e2 = norm(hypergeom(a,c,x)...
                -(q21+q22*x+q23*x.^2+q24*x.^3)./(1+d22*x+d23*x.^2),2)
    einf=norm(hypergeom(a,c,x)...
                -(q21+q22*x+q23*x.^2+q24*x.^3)./(1+d22*x+d23*x.^2),inf)

%% f=quartic/quadratic

% Set up matrix and vectors for linear system
    z1=zeros(7,1); B1=zeros(7,7); y1=zeros(7,1);

% Define and solve linear system
    tic, z1=[A(1),A(2),A(3),A(4),A(5),A(6),A(7)]';
    B1=[1,0,0,0,0,0,0;0,1,0,0,0,-A(1),0;0,0,1,0,0,-A(2),-A(1);0,0,0,1,0,...
-A(3),-A(2);0,0,0,0,1,-A(4),-A(3);0,0,0,0,0,-A(5),-A(4);0,0,0,0,0,-A(6),-A(5)];
    y1=B1\z1;

% Define terms in Pade approximant
    q31=y1(1)
    q32=y1(2)
    q33=y1(3)
    q34=y1(4)
    q35=y1(5)
    d32=y1(6)
    d33=y1(7)

% Compute Pade approximant and errors
    f3=@(x) (q31+q32*x+q33*x.^2+q34*x.^3+q35*x.^4)./(1+d32*x+d33*x.^2); toc

    e1 = norm(hypergeom(a,c,x)...
          -(q31+q32*x+q33*x.^2+q34*x.^3+q35*x.^4)./(1+d32*x+d33*x.^2),1)
    e2 = norm(hypergeom(a,c,x)...
          -(q31+q32*x+q33*x.^2+q34*x.^3+q35*x.^4)./(1+d32*x+d33*x.^2),2)
    einf=norm(hypergeom(a,c,x)...
          -(q31+q32*x+q33*x.^2+q34*x.^3+q35*x.^4)./(1+d32*x+d33*x.^2),inf)
```

References

[1] M. Abramowitz, I. A. Stegun: *Handbook of Mathematical Functions: with Formulas, Graphs and Mathematical Tables*, Wiley, 1972

[2] A. Bayliss, A. Class, B. J. Matkowsky: *Note: Roundoff Error in Computing Derivatives Using the Chebyshev Differentiation Matrix*, Journal of Computational Physics, **116**, pp.380-3, 1994

[3] C. W. Clenshaw: *A comparison of polynomial approximations with truncated Chebyshev series expansions*, SIAM Journal of Numerical Analysis, **1**, pp.26-37, 1964

[4] S. D. Fisher: *Best approximation by polynomials*, Journal of Approximation Theory, **21**, pp.43-59, 1977

[5] A. Gil, J. Segura, N. M. Temme: *Numerical Methods for Special Functions*, SIAM, 2007

[6] H. B. Keller: *Numerical Methods For Two-Point Boundary-Value Problems*, Dover Publications, 1992

[7] G. M. Phillips, P. J. Taylor: *Theory and Applications of Numerical Analysis*, Second Edition, Academic Press, 1996

[8] M. J. D. Powell: *Approximation theory and methods*, Cambridge University Press, 1991

[9] L. N. Trefethen: *Spectral Methods in Matlab*, SIAM, 2000