Publications of the Institute of Geophysics, Polish Academy of Sciences
Geophysical Data Bases, Processing and Instrumentation
DOI: 10.25171/InstGeoph_PAS_Publs-2022-046

Application of the MATLAB bvp4c Solver in the Linear Stability Analysis of Some Magnetohydrodynamic Problems

Marek J. GRĄDZKI

Institute of Geophysics, Polish Academy of Sciences, Warsaw, Poland

✉ mgradzki@igf.edu.pl

Abstract

We present our own ready-to-use MATLAB script that uses bvp4c, namely, the built-in MATLAB solver designed to solve boundary value problems with unknown parameters. The script is used to study the linear stability of an exemplary system of ordinary differential equations with appropriate boundary conditions, which have the form of the eigenvalue problem. These equations are classical equations of magnetohydrodynamics describing a layer of electrically conductive fluid in a magnetic field, in which, under the assumptions made, magnetorotational and magnetic buoyancy instabilities may occur, both associated with some important astrophysical phenomena. We present sample results where we successfully use the bvp4c solver to solve those equations and find eigenvectors and eigenvalues of the most unstable linear perturbations of the assumed basic state.

Keywords: magnetohydrodynamics, eigenvalue problem, MATLAB bvp4c, linear stability, instabilities.

1. INTRODUCTION

The 19th century was the heyday of classical physics, and thus also, closely related to it, mathematical methods of solving and analyzing differential equations. The end of this century resulted in fundamental works on the so-called qualitative theory of differential equations, by the great mathematicians of the time, such as Henri Poincaré or Aleksandr Lyapunov, who introduced the theory of stability of differential equations. In the 20th century, subsequent scientists developed this theory, specifying its mathematical foundations (Nemyckij and Stepanov 1989), analyzing its geometric aspects (Arnol'd 1992), using algebraic methods (Lanczos 1996) or, finally, going beyond the original assumptions and motivations, creating new interdisciplinary branch of science, such as the dynamical systems theory (Glendinning 1994). At the same time, the popularization and constant improvement of computing devices resulted in a rapid development of theories and applications of numerical methods for solving differential equations (Ackleh et al. 2010).

Although the last century was a time of breakthrough discoveries in completely new fields, such as quantum and relativistic physics, the classical physics continued to benefit from the development of analytical and numerical methods. One of its branches where new tools were successfully used was fluid mechanics. On the one hand, its foundations have been established and strictly mathematically described in many famous works (see e.g. Batchelor 2000; Lamb 1975), and on the other hand, it has been widely used to describe and study physical systems in many related fields, such as geophysics (Pedlosky 1987) or astrophysics (Chandrasekhar 1961). On their outskirts, a new branch of physics emerged, where some of their problems, together with methods and models of fluid mechanics and electromagnetism, resulted in a theory named by its founder, the Swedish physicist Hannes Alfvén, the magnetohydrodynamics (commonly abbreviated as MHD). Thanks to his work and that of his successors, a strict physical model of electrically conductive fluids under the influence of a magnetic field was created. It was successfully used to describe many phenomena related to the generation, maintenance and evolution of the magnetic field of planets, stars, accretion disks, or even entire galaxies, but also in engineering issues such as tokamaks, MHD motors or sensors (Roberts 1967; Moffat 1978).

One of the objects of interest of magnetohydrodynamics, in which the qualitative methods for differential equations are widely used, is the stability analysis of MHD systems. Although the description of fluid motion as a perturbation of a given basic state is well known from classical hydrodynamics (Drazin and Reid 2004), it turned out to be extremely useful in the case of electrically conductive fluids like plasma or liquid metals. The reason is the great variety of phenomena occurring in astrophysical objects that can be described as MHD waves and instabilities (Chandrasekhar 1961).

Among other things, it is worth mentioning two types of MHD instabilities. The first is associated with phenomena related to the extraction of plasma along with a strong magnetic field from the star interiors. These are, for example, prominences, coronal mass ejections, flares or star spots, well known due to observation of solar activity. It is believed that the responsible mechanism is the so-called magnetic buoyancy instability (MBI). It can occur when a toroidal (azimuthal) magnetic field decreases, at least locally, in the direction opposite to the gravity force (Acheson 1979; Gilman 1970; Hughes 1985). On the other hand, the complementary poloidal magnetic field component is a source of the so-called magnetorotational instability (MRI) if there is also a proper differential rotation of a conducting fluid in the radial direction. It occurs commonly in astrophysical systems and is considered to be particularly important in the case of accretion disk dynamics, where it is believed to be the main mechanism of angular momentum transfer necessary for the accretion phenomenon (Balbus and Hawley 1991).

From the mathematical point of view, it can be said that these instabilities are growing with time solutions of complicated systems of nonlinear magnetohydrodynamics equations. With the development of computers and numerical methods, a deeper theoretical studies of such systems have become possible. This applies both to the linear stability analysis and to the study of non-linear effects, usually impossible to analyze in any other way than by computer methods. The linear analysis usually involves numerical solving of the systems of differential equations that have been linearized due to linear perturbations, e.g. of the Fourier type (Chandrasekhar 1961). This is usually connected with verification of the difficult-to-obtain analytical results and is the first stage preceding the study of non-linear instabilities. Hence, the universality and necessity of this type of analysis.

The linear stability analysis can be done using a variety of computing environments designed to work on such problems. In this paper, we present one of such tools, namely, the

MATLAB program. It is commonly used to solve complicated differential equations, both ordinary (Keskin 2019; Shampine et al. 2003) and partial (Coleman 2013; Smith 1985). We discuss a special case of stability analysis, when a linearized system of equations can be represented as a boundary value problem (BVP), where the instability growth rate parameter plays the role of the eigenvalue of the system of equations. MATLAB is a convenient tool for dealing with such issues since it has a dedicated built-in solver called "bvp4c". We present its capabilities on the example of the MHD system in which the above-mentioned MBI and MRI instabilities are present. In the appendix, we attach our own ready-to-use MATLAB script in the form of a user-defined function named "mainBVPE", which, using this bvp4c solver, determines the eigenvectors and eigenvalues of the implemented system of ordinary differential equations.

The bvp4c is the main MATLAB routine for solving the boundary value problems. It uses the finite difference method: the three-stage Lobatto IIIa, a collocation fourth-order formula. The solver can be used to deal with systems of first-order differential equations with various complications, i.e. nonlinearities, complex-domain coefficients, singular terms, multipoint boundary conditions, and unknown parameters (eigenvalues). It is relatively easy to use and has a good documentation (for more details, see MATLAB built-in help, Kierzenka (2022), Shampine and Kierzenka (2001)). On the other hand, because of its algorithm, the bvp4c user has to provide the initial mesh and initial approximation of the solution at those mesh points. For many even complicated equations, this "initial guess" can be as simple as a constant function, but for other problems the result could be poor even for a relatively good input (e.g. for the complex functions or extreme values of the equation coefficients). Usually a good way to deal with this problem is to solve the equation under study multiple times in a loop, using the found solution as an initial guess for the calculation in the next step of the loop. This so-called "method of continuation" is a powerful tool worth using for difficult issues (for some examples, see Kierzenka (2022)). Moreover, from the physical point of view, the most troublesome issue could be the BVP with the unknown parameters to be determined, namely, the eigenvalue problem. The solver treats the eigenvalue as an additional unknown constant function so it also needs an additional boundary condition for the solution. This can be a problem if we are considering a real physical system and there is no justification for imposing an additional boundary condition on the solution we are looking for. Fortunately, there are usually ways around this problem and they are presented in this paper (see also Shampine et al. (2003)).

Due to the disadvantages described above, the bvp4c solver is not suitable for convenient and comprehensive analysis of the full spectrum of differential operators associated with the examined equations. In such situations, a better choice would be to use classical algebraic methods involving the study of matrices of differential linear operators of discretized equations (Lanczos 1996) or to use implemented MATLAB packages based on spectral methods (Trefethen 2000). However, bvp4c is an excellent tool for searching for separate eigenvalues of the analyzed BVP. This applies both to the ones close to the given initial values (e.g. to check the analytical results) and to the search for the most unstable solutions, i.e. eigenvectors associated with the greatest eigenvalues. The latter issue is particularly important in the study of the linear stability analysis of the real physical systems, because the most linearly unstable perturbations dominate the system dynamics after a sufficiently long time, at least before non-linear effects start to play an important role (see Mizerski et al. (2013) on the example of magnetic buoyancy instability). Therefore, in this article we present in practice the possibility of using the bvp4c solver to find the most unstable solutions of a given system of MHD equations, about which we know from the theory that they can be unstable due to the previously-mentioned magneto-rotational and magnetic buoyancy instabilities.

The paper is organized as follows: in Section 2 we present an exemplary MHD system whose linear stability we want to investigate using the MATLAB bvp4c solver. The MATLAB script that uses this routine is described in detail in Section 3 and its ready-to-use source code can be found in the Appendix. Section 4 contains sample results obtained by using this script, and the final Section 5 briefly summarizes all the work.

2. EXEMPLARY PHYSICAL PROBLEM

In this section, we describe an example of a physical problem that can be studied using the method announced in Section 1. Namely, we present here the theoretical introduction to the analysis of the linear stability of some MHD system. This system is related to simplified local models of astrophysical phenomena such as plasma instabilities and magnetic field generation in the interiors of stars (cf. Acheson 1979; Balbus and Hawley 1991; Gilman 1970, 2018a,b; Gilman and Dikpati 2014; Gradzki and Mizerski 2018; Hughes 1985; Mizerski et al. 2013). Since the purpose of this article is to present the application of the MATLAB bvp4c solver, in this section we only derive the main ODE which we solved by the MATLAB script presented in the Appendix.

Let's consider the fluid (solar plasma for example) which is compressible, inviscid, isothermal and electrically perfectly conducting, described by the ideal gas law and located between the two parallel infinite planes. Assuming the Cartesian coordinate system, those planes are determined by the equations $z = 0$ and $z = d$. We take into account the constant rotation $\boldsymbol{\Omega} = (0, \Omega, 0)$ and the constant gravity $\boldsymbol{g} = (0, 0, -g)$. The latter vector determines the vertical direction in the geometry of our case.

We adopt the standard MHD equations to describe such a system (cf. Moffat 1978; Roberts 1967; Mizerski et al. 2013). We then nondimensionalize them as follows: layer width d is the unit of length, free fall time $\sqrt{d/g}$ is the unit of time, free fall speed $\sqrt{gd}$ is the unit of velocity in the *y*- and *z*-direction, characteristic magnitude of the shear flow U^* is the unit of velocity in the *x*-direction; the values of the magnetic field B^*, pressure p^*, and density ρ^* at the top of the fluid layer $(z = 1)$ are the units of those quantities; the T^* is the unit of the constant temperature of the system.

Hence the governing MHD equations (successively: Navier-Stokes equation, the continuity equation, the magnetic induction equation, and the ideal gas law equation) for the main physical quantities (velocity $\boldsymbol{u}$, magnetic field $\boldsymbol{B}$, density $\bar{\rho}$, and pressure $\bar{p}$) take the following nondimensional forms:

$$\bar{\rho}\left(\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u}\cdot\nabla\boldsymbol{u}\right) = -\nabla\left(P\bar{p} + L\frac{\boldsymbol{B}^2}{2}\right) + L\boldsymbol{B}\cdot\nabla\boldsymbol{B} - U_o\bar{\rho}\,\boldsymbol{e_z}\times\boldsymbol{u} - \bar{\rho}\,\boldsymbol{e_z}, \tag{1a}$$

$$\frac{\partial\bar{\rho}}{\partial t} + \nabla\cdot(\bar{\rho}\,\boldsymbol{u}) = 0, \tag{1b}$$

$$\frac{\partial \boldsymbol{B}}{\partial t} + \boldsymbol{u}\cdot\nabla\boldsymbol{B} = \boldsymbol{B}\cdot\nabla\boldsymbol{u} - \boldsymbol{B}(\nabla\cdot\boldsymbol{u}), \tag{1c}$$

$$\bar{p} = a\bar{\rho}, \tag{1d}$$

with the following nondimensional parameters:

$$L = (B^*)^2/(\mu_0\rho^* gd), \qquad P = p^*/(\rho^* gd), \qquad \alpha = R\rho^* T^*/p^*, \tag{2a}$$

$$U_o = 2\Omega d/\sqrt{gd}, \qquad U_u = U^*/\sqrt{gd}, \tag{2b}$$

where R is the ideal gas constant, μ_0 is the vacuum permeability, L is the squared nondimensional Alfven speed, product $P\alpha$ is the squared nondimensional isothermal speed of sound, U_o is the nondimensional rate of rotation, and U_u is the nondimensional speed of characteristic shear flow. The nondimensional fluid velocity $\boldsymbol{u}$ and magnetic field $\boldsymbol{B}$ have the following components:

$$\boldsymbol{u} = [U_u\bar{u}, \bar{v}, \bar{w}], \tag{3}$$

$$\boldsymbol{B} = [\bar{a}, \bar{b}, \bar{c}], \tag{4}$$

and in general they are all functions of spatial coordinates x, y, z, and time t. The same applies to pressure $\bar{p}$ and density $\bar{\rho}$.

Let's define the equilibrium basic state of the fluid by the horizontal z-dependent magnetic field in the form $\boldsymbol{B_0}(z) = (a_0(z), b_0, 0)$ and the horizontal z-dependent shear flow $\boldsymbol{u_0}(z) = (u_0(z), 0, 0)$. Hence, from the z-component of the Navier-Stokes and ideal gas law equations, we get the equation for the basic state density $\rho_0(z)$:

$$\frac{d}{dz}(P\alpha\rho_0(z) + La_0^2(z)/2) = -\rho_0(z)(f - U_oU_uu_0(z)), \tag{5}$$

with the nondimensional boundary condition $\rho_0(z = 1) = 1$. The parameter f is the indicator of the gravity g (which is equal to 1 after our nondimensionalization), namely $f = 1$ if the gravity is present (which we always assume in this paper).

The next step in the linear stability analysis of the system is to choose the form of the perturbations of the assumed basic state. The system is homogenous along the x and y axis and hence the perturbation can adopt the form of a Fourier modes varying in those directions. In this analysis we focus on a simpler case, the so-called "interchange modes", which are "two-dimensional" and "axisymmetric" from the rotational point of view, i.e. they vary in the y-direction and do not depend on the x argument. The stability analysis of the full "three-dimensional" modes is also possible with this method, but the calculations are far more laborious (cf. Gradzki and Mizerski 2018; Mizerski et al. 2013). Hence the perturbations of the velocity, magnetic field, pressure, and density take the forms:

$$[u(z), v(z), w(z)]e^{st+ik_yy}, \qquad [a(z), b(z), c(z)]e^{st+ik_yy}, \tag{6a}$$

$$p(z)e^{st+ik_yy}, \qquad \rho(z)e^{st+ik_yy}, \tag{6b}$$

where s and k_y are the perturbation growth rate and the wavenumber, respectively.

To investigate the linear stability of the system due to these perturbations, we introduce them into the system of Eqs. (1a)–(1d) including the basic state Eq. (5) and neglect the non-linear terms. We then obtain the system of eight linear ordinary differential equations for eight unknown amplitudes of the perturbations: $u(z)$, $v(z)$, $w(z)$, $a(z)$, $b(z)$, $c(z)$, $p(z)$, and $\rho(z)$. The coefficients of those ODE's contain all z-dependent basic state functions, non-dimensional parameters (2a)–(2b), growth rate s, and wavenumber k_y. On the other hand, since the wavenumber k_y can take any value (positive and real, for the sake of simplicity), the growth rate s can be considered as the eigenvalue of the linear differential operator given by the system of equations, for which it has the solution – an associated eigenvector of the amplitudes.

As we know from the linear stability theory (cf. Chandrasekhar 1961; Drazin and Reid 2004) our system is unstable due to assumed perturbations (6a)–(6b) if we can find any eigenvectors exponentially growing over time, that is, the ones with the positive real part of their growth rates (eigenvalues), i.e. with $\Re(s) > 0$. On the other hand, from Balbus and Hawley (1991), Gilman (2018a,b), and Mizerski et al. (2013), we can expect in our MHD system three types of instabilities, i.e., magnetorotational instability, magnetic buoyancy instability, and (non-magnetic) centrifugal instability. Hence it is an interesting physical system to study its linear stability.

Obviously the resulting system of equations for the z-dependent perturbations amplitudes can be solved only by numerical methods. As it was mentioned in Section 1, there are various approaches to these types of issues with MATLAB program (cf. Keskin 2019; Shampine et al. 2003; Trefethen 2000). If one wants to use a built-in bvp4c solver, the problem is the need to reduce the equations of the system to a form containing the derivatives of each unknown function on only one side of the equation, namely in the algebraic form, $\frac{d\boldsymbol{f}(z)}{dz} = \widehat{M}(z)\boldsymbol{f}(z)$, where $\widehat{M}(z)$ is a matrix of the system coefficients that do not contains the differential operators. This is often not an easy task and the system presented in this section is an example of such a situation. However, it is usually only a matter of tedious calculations that lead to very complicated coefficients of the equations, which take on enormous proportions when there is diffusion in the system (cf. Gradzki and Mizerski 2018). In such cases, the use of symbolic calculation programs such as Mathematica becomes invaluable.

In our case, we can with some effort transform the derived system of eight ODE's for eight amplitudes of the perturbations (u, v, w, a, b, c, p, and ρ) to just one second-order linear ordinary differential equation for the amplitude of the vertical velocity perturbation $w(z)$:

$$W_2(z)\frac{d^2w(z)}{dz^2} + W_1(z)\frac{dw(z)}{dz} + W_0(z)w(z) = 0, \tag{7}$$

where the z-dependent coefficients of Eq. (7) are very complicated and can be found in the body of the "odefun" function in the Appendix (Section "Nested functions").

Now we have to impose the boundary conditions on the function $w(z)$. Since there is no diffusion in our system, the simplest choice is the impermeability condition at the horizontal boundaries of our fluid layer, namely:

$$w(z=0) = 0, \qquad w(z=1) = 0. \tag{8}$$

In conclusion, the linear stability analysis of our MHD problem reduces to a two-point boundary value problem (BVP): Eq. (7) with boundary conditions (8) and the eigenvalue s (the growth rate of the perturbation introduced in Eqs. (6a)–(6b)). Those equations can be obviously transformed into the system of first-order equations with proper BC's:

$$\frac{dw_1(z)}{dz} = w_2(z), \tag{9a}$$

$$\frac{dw_2(z)}{dz} = -\left(\frac{W_1(z)}{W_2(z)}\right)w_2(z) - \left(\frac{W_0(z)}{W_2(z)}\right)w_1(z), \tag{9b}$$

$$w_1(z=0) = 0, \qquad w_1(z=1) = 0, \tag{9c}$$

where $w_1(z) = w(z)$ and $w_2(z) = dw(z)/dz$. This is the form required by the bvp4c solver.

The z-dependent coefficients of Eq. (9b) contain all basic state functions, nondimensional parameters, the wavenumber k_y (whose value we choose for each calculation) and the growth rate s (which we want to determine for the selected k_y). Fortunately, the form of those coefficients is not a problem for the solver like the bvp4c and the system of Eqs. (9a)–(9c) can be solved with this routine. The numerical procedure using this method is presented in the next Section 3 and the source code of the ready-to-use MATLAB script solving this problem can be found in the Appendix.

3. NUMERICAL PROCEDURE

The MATLAB built-in solver bvp4c is a great tool to deal with problems like the boundary value problem given by the system of Eqs. (9a)–(9c). However, using it requires some practice and the tasks to be solved require some preliminary preparation, especially if we are dealing with eigenvalue problems. In this section, we present the MATLAB script that uses bvp4c to solve our BVP (source code in the Appendix) and on this occasion we discuss the most important issues regarding the efficient use of the solver.

The script was created with MATLAB R2018a version. It has a form of a void function and hence it can be executed from the MATLAB command window with its name, namely “mainEBVP”, after copying and saving it to a file named mainEBVP.m. It means that all input data should be changed inside the main function, since it has no arguments.

The source code is divided into the following types of appropriately named sections: “General information”, INPUT, SOLVER, OUTPUT, and “Nested functions”. The first section contains a brief mathematical and physical description of the MHD linear stability problem presented in the previous chapter, as well as the specification of the mainEBVP function itself. Since that part is self-explaining, we start the description of the script from the last section, because it contains mainly the key functions used directly by the bvp4c solver in the SOLVER sections of the program. Then we present other sections from the main part of the mainEBVP user-defined function.

3.1 Nested functions

The Section “Nested functions” at the end of the script contains mainly all the functions that are required for the solver to work. Handles to those functions are used by the MATLAB built-in routines: bvp4c (solves the BVP equation), bvpset (sets the solver options), and bvpinit (creates the initial guess for the solution); see MATLAB help for more info about those functions.

3.1.1 Function y_guess_01

This nested function is used by the bvpinit routine to create the initial guess for the solution in the form of a constant function (see code section no. 06). This can work surprisingly well even for relatively complicated equations. It is worth noting that the constant guess does not need to satisfy the boundary conditions in a nested function bcfun or any part of the system of equations in the nested function odefun.

3.1.2 Function y_guess_02

This is a variant of the previous function, but here the initial guess is in the sine-type form. This is usually a good choice if we expect a solution of a similar character. On the other hand, the user can define here, in a similar manner, any function that he expects to work.

3.1.3 Function odefun

This is the key nested function of the script, in which the main system of equations to be solved by the solver is given. It has implemented the complicated z-dependent coefficients W_0, W_1, and W_2 of Eq. (9b) which contain all input parameters and basic state functions from the INPUT sections of the code, as well as the unknown eigenvalue s. The function also needs the numerical solution of Eq. (5) for the basic state density $\rho_0(z)$ (denoted as *r0* in the script) which is solved numerically at the beginning of the code section no. 07. The handle to the odefun function is a necessary argument of the bvp4c routine.

Now, since our analysis is linear (in other words: perturbations are infinitesimally small compared to the basic state) and the corresponding BVP is an eigenvalue problem, the solution $w(z)$ is found only up to the unknown constant factor which cannot be determined in a linear analysis. On the other hand, from the numerical method point of view, the presence of an eigenvalue in the equation forces the user to impose an additional boundary condition to determine the solution uniquely (cf. Shampine and Kierzenka 2001). This can be a problem if we do not have a physical reason for such an extension. Especially in our case, there is no justification for the additional BC on the derivative of the function $w(z)$, since the fluid is assumed to be inviscid. Nevertheless, to use the bvp4c solver, we would need a third boundary condition for the system of Eqs. (9a)–(9c).

This issue can by overcome in the following way: to our system of equations we can add an additional one for the new function $w_3(z)$ which can be used as a normalization of our "main" solution $w_1(z)$ by the proper formulation and by using two additional boundary conditions for $w_3(z)$ (one for new equation and one for the eigenvalue determination). The simplest choice is to define function $w_3(z)$ as an antiderivative of the $w_1(z)$ and the additional BC's in a form that defines the integral of the latter function as equal to unity (or any other constant number). In such a case, our BVP given by the system of Eqs. (9a)–(9c) takes the following new form:

$$\frac{dw_1(z)}{dz} = w_2(z), \tag{10a}$$

$$\frac{dw_2(z)}{dz} = -\left(\frac{W_1(z)}{W_2(z)}\right) w_2(z) - \left(\frac{W_0(z)}{W_2(z)}\right) w_1(z), \tag{10b}$$

$$\frac{dw_3(z)}{dz} = w_1(z), \tag{10c}$$

$$w_1(z=0) = 0, \qquad w_1(z=1) = 0, \tag{10d}$$

$$w_3(z=0) = 0, \qquad w_3(z=1) = 1. \tag{10e}$$

The normalization of the function $w_1(z)$ via Eq. (10c) can be done in many other ways. Probably the most elegant and safest option from a mathematical point of view is to normalize $w_1(z)$ to a square-integrable function, namely define Eq. (10c) as:

$$\frac{dw_3(z)}{dz} = |w_1(z)|^2, \tag{11}$$

to be sure that we do not miss any solution. This option seems reasonable, especially if we expect eigenvector $w(z)$ to be a complex function.

The above-described procedure for extending the initial system of Eqs. (9a)–(9c) allows us to use the bvp4c solver. Of course, this method has one major disadvantage: by adding an additional equation to our BVP, we increase the number of calculations, and therefore the resource consumption and time needed to obtain the result. The greater the increase, the more complicated the additional normalizing equation. However, this is probably the most reasonable approach to solving the eigenvalue problems with bvp4c solver from both a mathematical and physical point of view.

3.1.4 Function bcfun

The handle to this function is another necessary argument of the bvp4c routine. It contains the boundary conditions implemented in the nested function odefun, namely Eqs. (10a)–(10e). As written above, in the case of the eigenvalue problem of three ODE's, the solver needs four BC's. In our case, these are two physical ones and two additional normalization conditions.

3.1.5 Function bcjac

This is the function that contains a Jacobian matrix of the boundary condition implemented in the function bcfun. It is a feature that can be enabled optionally via the in-built bvpset routine. If this option is turned on, the bvp4c solver will not need to numerically approximate partial derivatives of the boundary conditions equations. This should at least slightly speed up the calculations. In practice, for typical BC's, the gain is not great, but, on the other hand, the implementation of this function is very simple.

The bvp4c solver also allows user to optionally use a Jacobian matrix for the main system of equation to be solved. Enabling this FJacobian option, in contrast to the Jacobian of boundary conditions, can speed up the calculations tremendously. Thus, it is recommended whenever the analytic form of the Jacobian is easy to find and implement. Unfortunately, for our BVP, this is practically impossible due to the huge complexity of the coefficients of Eq. (10b).

3.1.6 Function extra_space

The last nested function is not related to the operation of the solver, but only to printing data on the screen and saving it to a file. It allows to print large numbers in a readable form, where thousands are separated by a space, e.g. extra_space(1e4) = 10 000 etc.

3.2 INPUT sections

The INPUT part of the script of the function mainEBVP consists of five sections numbered from 01 to 05. As the name suggests, this is the only part of the code where the user provides input data to solve the boundary value problem implemented in the nested functions odefun and bcfun described above. It consists of the following code blocks:

3.2.1 "01: INPUT: Main parameters"

In this section the user has to choose values of all physical parameters characterizing the considered system, namely the nondimensional constants (2a)–(2b), the quantities related to the base state functions $a_0(z)$, $b_0(z)$, $u_0(z)$, $\rho_0(z)$, and the wavenumber k_y (see Section 2 for details).

3.2.2 "02: INPUT: Basic state functions"

The form of the basic state toroidal magnetic field $a_0(z)$ and shear flow $u_0(z)$ should be chosen in this section. The attached version of the script contains the simplest linear forms of those functions and the optional quadratic form of $u_0(z)$. If one needs to "switch off" the magnetic field or the shear flow at the time of calculation, the safest would be to select the corresponding zero function in this section and "comment" or delete the others. There are also no obstacles for the user to define his own basic state functions of any form.

3.2.3 "03: INPUT: Solver options"

The next section includes the ability to select all implemented bvp4c options through the function bvpset. They are all well described in the in-built MATLAB help. From the scientific point of view, the most important option is probably the RelTol, namely, the relative error tolerance. Its default value is 10^{-3} which corresponds to 0.1% accuracy; however, the authors of the solver recommend a value of 10^{-5} for the scientific calculations (see Shampine and Kierzenka 2001). Decreasing of the RelTol parameter will be necessary if the user wants to get the eigenvalue s with greater accuracy. Of course, with increasing the accuracy of the calculations, their duration also increases, sometimes very significantly, especially for the poor initial guesses for the eigenvalue. A fairly effective way to deal with this problem is to use the "method of continuation" described below in Section 3.2.5, when we can increase the accuracy in every step of the loop (cf. also Shampine et al. 2003).

3.2.4 "04: INPUT: Initial guess properties"

This section is probably the most important for the successful completion of the calculations and their duration. Here the user has to choose an initial guess for both the eigenvalue s and the solution $w(z)$. The value of the former should be as close as possible to the expected value (if it is known, for example, from analytical considerations), and the solver should calculate the eigenvalue closest to the input value (which is not always true, especially if it is a complex number). The latter one is selected from the possibilities implemented in the nested functions described in the previous Section 3.1, i.e. "y_guess_01" or "y_guess_02". In addition, one has to select the number of starting mesh points. Although the bvp4c solver selects optimally the final mesh during its internal process, this initial mesh size has a huge impact on how long the calculation takes. In general, it is worth starting with not very dense meshes (of the order of several dozen points or even a dozen or so points) and densify them as necessary. On the other hand, it is better to start with a grid with more points (hundreds or even thousands) if we expect strongly oscillating or boundary layer type solutions.

3.2.5 "05: INPUT: OPTIONAL: solving in the loop"

The last section of this part of the code contains all input parameters necessary to use the "method of continuation" mentioned before. The user has to select a value for the loop_switch variable: "on" (if one wants to use the method) or "off"' (if not).

This method offers various possibilities. The simplest one requires the "off" value of the variable loop_over_ky and consists of repeated use of the bvp4c solver in subsequent steps of the loop without changing the parameter input values. In this case, the solution obtained in a given step becomes the initial guess in the next one. This often allows to get a solution even with a poor initial guess of the function we are looking for. However, one has to be careful when calculating the eigenvalue problems, because the solver in subsequent steps can "stick" to the eigenvalue (and eigenvector as well) found in the first step. This may produce undesirable results depending on user expectations, e.g. if we are looking for the most unstable solution.

The second option offered by the "continuation method" is to find solutions in a loop for such extreme values of equation parameters that it would not be efficient (or even possible) in a single calculation. As an example, in the version of the script from the Appendix, the possibility of increasing the wavenumber k_y value in subsequent loop steps has been implemented. This requires the "on" value of the variable loop_over_ky. If the user wanted to solve Eqs. (10a)–(10e) for, say, $k_y{\sim}10^6$ and the remaining parameters of unity order, this would be a big challenge for the solver, requiring a perfect initial guess. The solution to this problem is the discussed method and starting calculations for a much smaller value of k_y, which will not cause difficulties for the bvp4c routine. Then we can increase this value in subsequent steps of the loop until the desired result is achieved. This method can give great results, but the increase in the value of the changed parameter must be optimally selected. If it is too low, the calculations will last unnecessarily long. If it is too big, the solver may not be able to handle the calculations. The use of this method, therefore, requires some practice and usually a certain number of tests.

The last possibility we mention here is to increase the accuracy of the calculation in each step of the loop. This is only partially implemented in the mainEBVP function script and requires "uncommenting" corresponding lines in the loop structure in block of the code no. 08. However, the idea is simple: calculations start with the standard value of the $\mathrm{RelTol} = 10^{-3}$ parameter (see Section 3.2.3) and are decremented in each step, e.g. by dividing by a factor of 10. Thanks to this, we can obtain the usually desired accuracy of the result.

Finally, it should be added that, in this section of code, the user can choose how many steps of the loop the data will be saved on the screen, in the file and on the plot.

3.3 SOLVER sections

In the next five blocks of the code, numbered from 05 to 10, all the calculations and instructions needed to obtain and present the results are done. The script sequentially performs the following tasks based on the input data: it creates the initial guess, solves the basic state Eq. (5) using ode45 routine, solves the main system of Eqs. (10a)–(10c) for boundary conditions (10d)–(10e) with bvp4c solver, optionally does calculations in a loop "by continuation", performs some additional calculations (like integrating the solution) and finally creates the string variables needed to print and save the output results. This part of code is well described in the comments, but the user does not need to study and change it, unless he wants to make serious modifications to the script.

3.4 OUTPUT sections

Code blocks of the function mainEBVP numbered 11 through 13 are responsible for presenting the output data. There are three types in the attached version of the script.

3.4.1 MATLAB command window

The script displays in the MATLAB command window all the most important input and output data, such as the values of: start and end date and time of calculation, some main equation parameters (in the current version: k_y and b_0), initial and final eigenvalue s, the argument for the largest value for the solution modulus, integrals of the solution (to check the given normalization condition), start and final options and parameters of the bvp4c solver operation**.**

Optionally, with the “continuation method” enabled (see Section 3.2.5), the basic results for calculation steps are printed on the screen at the frequency selected by the user in code block no. 06.

3.4.2 Data files

Two text files are generated by default, if they do not exist yet. The first one, called mainEBVP_log.txt, is a log file, where for subsequent executions of the program, the same data that the user sees in the MATLAB command window is saved. New results are appended to the end of the file. Thanks to the data from the file, the user can, for example, analyse the duration of calculations depending on the input solver settings. Note, however, that the calculated eigenvalue s is saved in the file, but not the solution $w(z)$. The latter one is saved in the second created file named mainEBVP_last_sol.txt, where the data is overwritten after each successful completion of the calculation. This file contains the final mesh selected by the solver and all components of the solution vector $w(z)$. In addition, it also contains the same data that is printed in the MATLAB command window.

Additionally, if the “continuation method” is used, the file loop_sol.txt is created. The basic results for the subsequent steps are saved to that file with the frequency chosen by the user in code block no. 06.

3.4.3 Plots

The version of the script attached to the article produces five graphs. The first is the plot of the basic state density $\rho_0(z)$ calculated from Eq. (5) at the beginning of code block no. 07.

The other graphs represent the solution found, namely, the eigenvector $w(z)$, which is the first component of the complete solution vector, denoted in the script as $w_1(z)$ (see Eqs. (7) and (9a)–(9c)). They are the separate plots of: real part of the solution $\Re\big(w(z)\big)$, its imaginary part $\Im\big(w(z)\big)$, its modulus $|w(z)|$, and all those three functions on one figure. All plots of $w(z)$ are normalized by dividing the values of the considered part of the function by the largest value of the modulus of that part. Of course, in case the user expects only purely real eigenvectors, it is most convenient to keep only the plot of $\Re\big(w(z)\big)$ and “comment out” the rest.

Optionally, with the “continuation method” enabled, an additional plot is created with subsequent solutions drawn at the frequency selected by the user in code block no. 06.

All plots created by the mainEBVP function are also saved in three types of files: MATLAB FIG, JPEG, and PDF.

4. EXEMPLARY RESULTS

As it was said in Section 1, the function mainEBVP included in the Appendix can be used to investigate the linear stability of the system under study in various ways. In this chapter, we present sample results: eigenvalues s and eigenvectors $w(z)$ (denoted also as $w_1(z)$, cf. Section 3.1.3) found for some fixed values of all parameters of the system. They are divided into three cases: purely real solutions, complex solutions, and solutions obtained by the "continuation method". In all three cases, the script solves the system of Eqs. (10a)–(10c) with boundary conditions (10d)–(10e) and selected forms of the basic state functions: toroidal magnetic field $a_0(z)$ and horizontal shear flow $u_0(z)$.

4.1 Real solutions

In this section we present exemplary results for the case where the found eigenvalues s and eigenvectors $w(z)$ have values in the real domain. Such a situation is taking place when we consider the tested system with the following basic state:

$$a_0(z) = 0, \tag{12a}$$

$$u_0(z) = \zeta z, \tag{12b}$$

hence in the absence of the toroidal magnetic field $a_0(z)$ and for the presence of the linear shear flow $u_0(z)$. The nondimensional parameter ζ is in the script of the mainEBVP function denoted as dz. To implement the appropriate basic state, the user has to select these functions in code block no. 02 in the script and set the variable la = 0.

From previous studies, it is known that such a system can be linearly unstable due to two types of instabilities, namely magnetorotational and centrifugal (see Balbus and Hawley 1991; Drazin and Reid 2004; Gilman 2018a,b). To check this for the given set of main system parameters, it is enough to find at least one unstable solution, i.e. one with a positive real part of the growth rate s (which is our eigenvalue). As an example, we assume the following values of the main parameters introduced in Section 2 as well as the value of the wavenumber k_y and a shear flow gradient ζ:

$$L = 0.25, \qquad P = 1.5, \qquad \alpha = 1.0, \tag{13a}$$

$$U_o = 0.9, \qquad U_u = 1.0, \qquad b_0 = 0.01, \tag{13b}$$

$$k_y = 150, \qquad \zeta = -1.0. \tag{13c}$$

The mainEBVP function was used to search for the unstable solutions for such settings and with the bvp4c solver accuracy parameter $\text{RelTol} = 10^{-5}$. The number of points of the initial mesh (the variable meshPointsNumber) is equal to 21 on the domain $[0,1]$ and the initial guess for the solution is a constant function (solinitNumber = '01'). There is no calculating in the loop (loop_switch = 'off').

According to Mizerski et al. (2013), taking into account the form of the Eq. (10b) coefficients and the assumed values of the system parameters, we can expect that the eigenvalues will be purely real numbers with an absolute value of the order of unity, i.e.: $|s| \sim 1$. In this situation, because the bvp4c solver requires an initial value for the unknown parameter s, it was necessary to carry out preliminary simulations in order to find by the "trial and error" method the region of possible positive eigenvalues. It turns out that such values exist, and therefore the considered system is linearly unstable due to perturbations assumed in the form (6a)–(6b).

We also managed to identify the most unstable mode, i.e. the eigenvector $w(z)$ associated with the highest possible eigenvalue s, as well as subsequent elements of the spectrum of linear operator associated with the equation under study. Table 1 contains the eight greatest eigenvalues s found for the values of parameters (13a)–(13c), and Fig. 1 presents their corresponding eigenvectors $w(z)$, normalized on the plot so that max(|w|) = 1.

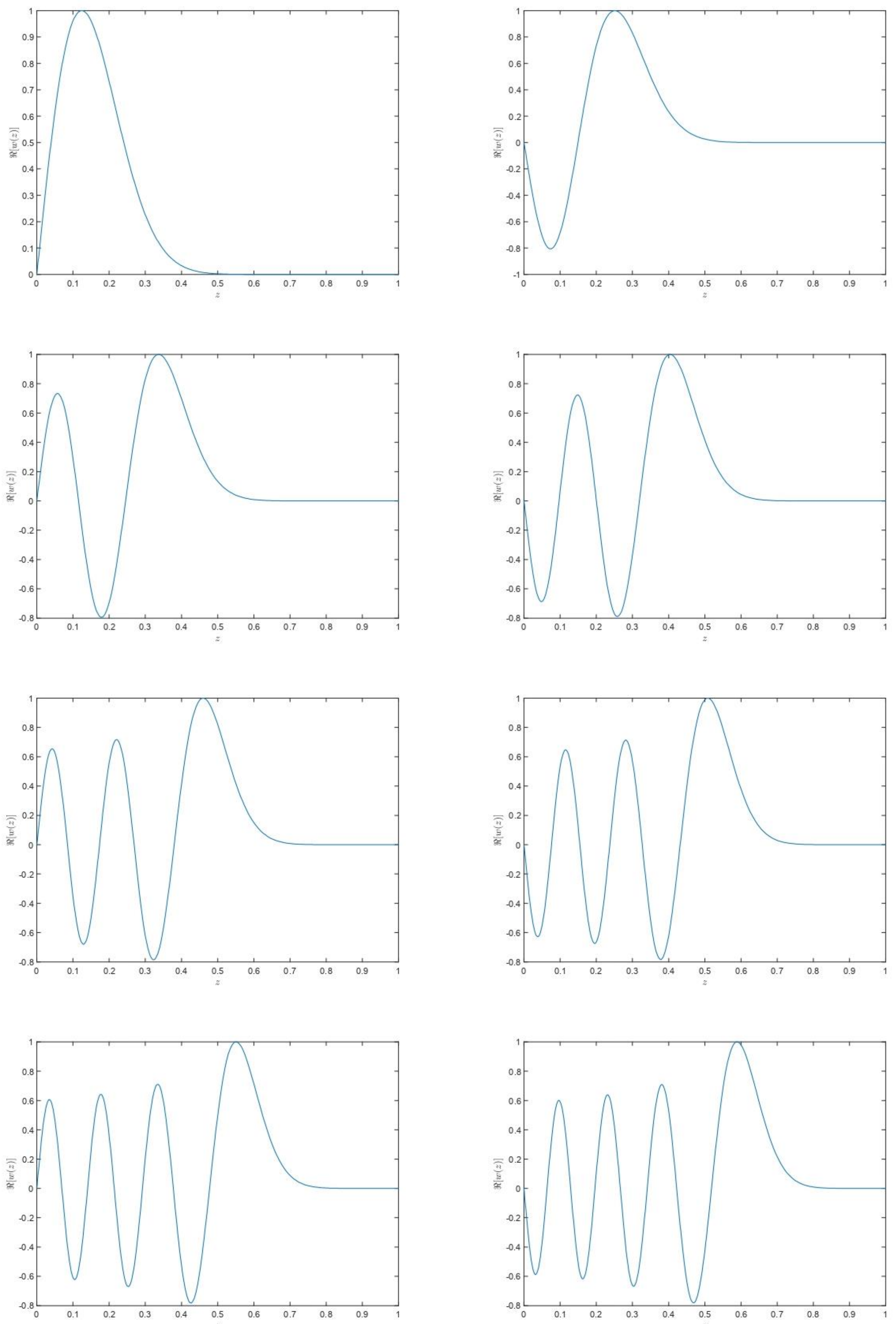

Fig. 1. The eigenvectors $w(z)$ associated with the eight greatest eigenvalues s of Eq. (7) for the values of the parameters (13a)–(13c) and basic state (12a)–(12b), found with the user-defined MATLAB function mainEBVP using the build-in bvp4c solver. The top left plot is related to the eigenvalue s_1 from Table 1, the top right is related to s_2 and so forth. The plots are normalized so that $\max(|w|) = 1$.

Table 1

The first eight greatest eigenvalues s of Eq. (7) for the values of the parameters (13a)–(13c) and basic state (12a)–(12b), found with the user-defined MATLAB function mainEBVP using the build-in bvp4c solver

s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8
0.49721	0.49412	0.49091	0.48758	0.48414	0.48058	0.47693	0.47317

4.2 Complex solutions

In this section we present exemplary results for the case with the complex values of the eigenvalues s and eigenvectors $w(z)$. Such a situation is taking place when we consider the system under study with the basic state in the following form:

$$a_0(z) = 1 + \lambda(1 - z), \tag{14a}$$

$$u_0(z) = \zeta z, \tag{14b}$$

hence, for the linear toroidal magnetic field $a_0(z)$ and shear flow $u_0(z)$. The nondimensional parameter λ is in the script of the mainEBVP function denoted as "la". To implement the appropriate basic state, the user has to select these functions in code block no. 02 in the script.

From previous studies, it is known that such a system can be linearly unstable due to three types of instabilities, namely magnetorotational, centrifugal and magnetic buoyancy instability (see Balbus and Hawley 1991; Drazin and Reid 2004; Gilman 2018a,b; Mizerski et al. 2013). To investigate this for the given values of main system parameters, it is enough to find at least one unstable solution, i.e. one with a positive real part of the growth rate s. As an example, we assume the following values of main parameters, wavenumber k_y, and the basic state functions gradients ζ and λ:

$$L = 0.25, \qquad P = 1.5, \qquad \alpha = 1.0, \tag{15a}$$

$$U_o = 0.9, \qquad U_u = 1.0, \qquad b_0 = 0.01, \tag{15b}$$

$$k_y = 150, \qquad \zeta = -1.0, \qquad \lambda = 1.4. \tag{15c}$$

Thus, this case can be treated as an examination of the influence of the toroidal z-dependent magnetic field $a_0(z)$ on the case presented in Section 4.1, for the perturbation with particular wavenumber k_y.

As in the previous section, the mainEBVP function was used to search for the unstable solutions for such settings and with the bvp4c solver accuracy parameter RelTol $= 10^{-5}$. The number of points of the initial mesh (the variable meshPointsNumber) is equal to 21 on the domain $[0,1]$ and the initial guess for the solution is a constant function (solinitNumber = '01'). There was no calculation in the loop (loop_switch = 'off').

Since the coefficients of Eq. (7) are complex functions, in this case we can expect that the eigenvalues will be complex numbers with the real part absolute value of the order of unity, i.e.: $|\Re(s)| \sim 1$. In this situation, because the bvp4c solver requires an initial value for the unknown parameter s, it was necessary to carry out preliminary simulations in order to find by the "trial and error" method the region of possible positive eigenvalues. In the complex case, this is more difficult than in the real one, since we have to select initial guess for both real and imaginary parts of the eigenvalue. However, it turns out that such eigenvalues exist, and therefore the considered system is linearly unstable due to the assumed perturbations.

We also managed to identify the most unstable mode, i.e. the eigenvector $w(z)$ associated with the highest possible real part of the eigenvalue s, as well as some subsequent elements of

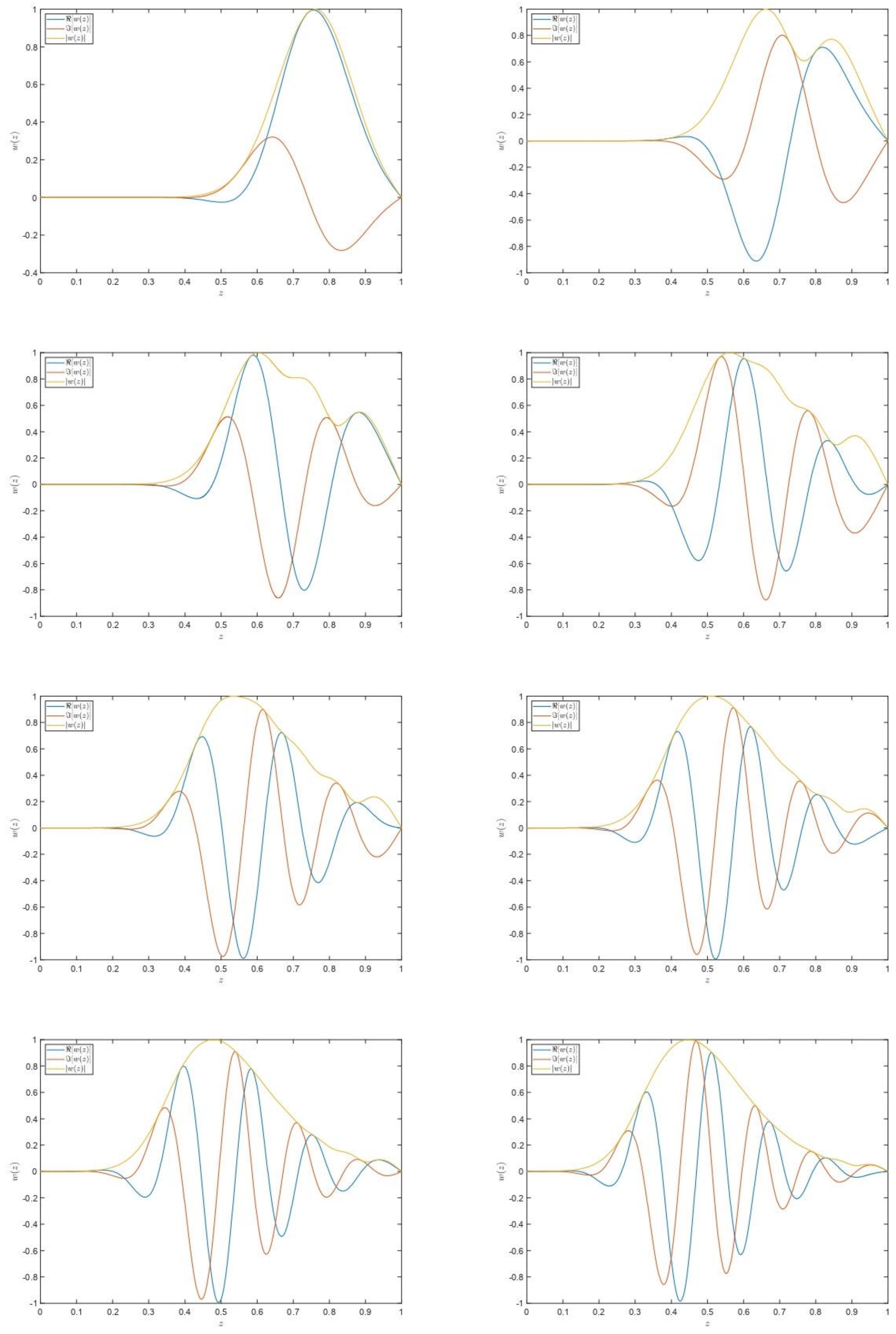

Fig. 2. The eigenvectors $w(z)$ associated with the eight eigenvalues s with the greatest real parts, for Eq. (7) for the values of the parameters (15a)–(15c) and basic state (14a)–(14b), found with the user-defined MATLAB function mainEBVP using the build-in bvp4c solver. The real part $\Re(w(z))$ is marked with a blue line, the imaginary part $\Im(w(z))$ with a red line, and modulus $|w(z)|$ with a yellow line. The top left plot is related to the eigenvalue s_1 from Table 2, the top right is related to s_2, and so forth. The plots are normalized so that $\max(|w|) = 1$.

the spectrum of linear operator associated with the equation under study. Table 2 contains eight eigenvalues s with the greatest real parts, found for the values of parameters (15a)–(15c). Figure 2 presents modulus as well as real and imaginary parts of their corresponding eigenvectors $w(z)$. The functions are normalized on the plot, so that max(|w|) = 1.

Table 2

The real and imaginary parts of the first eight eigenvalues s with the greatest real parts of Eq. (7) for the values of the parameters (15a)–(15c) and basic state (14a)–(14b), found with the user-defined MATLAB function mainEBVP using the build-in bvp4c solver

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8
$\Re(s)$	0.57566	0.57253	0.56936	0.56615	0.56289	0.55963	0.55637	0.55313
$\Im(s)$	–0.27533	–0.27306	–0.27063	–0.26804	–0.26534	–0.26257	–0.25976	–0.25692

4.3 Method of continuation

In this section we present the exemplary results obtained using the "method of continuation" (for more details about the method, see Kierzenka (2022), Shampine et al. (2003)). As mentioned above, this method is based on a repeated use of the bvp4c solver in a loop, where in subsequent steps the solution found in the previous step is used as an initial guess for the next calculation. In this chapter, we show how to use this method to solve the eigenvalue problem for large values of certain parameters of the equation (in our case: for the large wavenumber k_y). In such situations, using the solver once can be very inefficient if we are unable to give an initial guess for the eigenvalue that is very close to the correct one. The answer to this problem is to find a solution for a value of the selected parameter comparable to the orders of magnitude of the other parameters (which is usually pretty easy even for poor initial guesses), and then increase this parameter in subsequent loop steps until the desired value. The key here is to choose the increment value so that, on the one hand, the solver finds subsequent solutions without any problems, and on the other hand, so that the calculations do not take too long. With large demanded values of the considered large parameter it may be necessary to gradually increase the increment, and therefore perform calculations in several loops.

To illustrate our example, we consider the system under study with the basic state in the following form:

$$a_0(z) = 1 + \lambda(1 - z), \tag{16a}$$

$$u_0(z) = 0, \tag{16b}$$

hence, in the presence of the toroidal magnetic field $a_0(z)$ and in the absence of the shear flow $u_0(z)$. To implement the appropriate basic state, the user has to select these functions in code block no. 02 in the script and set the variable $dz = 0$.

From previous studies, it is known that such a system can be linearly unstable due to magnetic buoyancy instability. We can also expect that the growth rate s (i.e. the eigenvalue) of a chosen eigenvector (e.g. the most unstable one) will increase and tend to a certain fixed value as the wavenumber k_y increases (Mizerski et al. 2013; Gradzki and Mizerski 2018). To investigate the stability of the system for the given values of main parameters, it is enough to find at least one unstable solution, i.e. one with a positive real part of the growth rate s. In this case, however, our goal will be to find the most unstable eigenvector for wavenumber $k_y = 100$, and then, using the "continuation method", determine its equivalent for wavenumber $k_y = 10^5$ along with its associated eigenvalue.

As an example, we assume the following values of the system parameters:

$$L = 0.25, \qquad P = 1.5, \qquad \alpha = 1.0, \tag{17a}$$

$$U_o = 0.0, \qquad U_u = 1.0, \qquad b_0 = 0.0, \tag{17b}$$

$$\zeta = 0.0, \qquad \lambda = 1.4. \tag{17c}$$

As in the previous section, the mainEBVP function was used to search for the unstable solutions for such settings and with the bvp4c solver accuracy parameter $\mathrm{RelTol} = 10^{-5}$. The number of points of the initial mesh (the variable meshPointsNumber) is equal to 21 on the domain $[0,1]$ and the initial guess for the solution is a constant function (solinitNumber = '01'). Since we want to do calculations in a loop with the k_y parameter increasing, we need to set the following values of the corresponding variables: loop_switch = 'on' and loop_over_ky = 'on'.

According to Mizerski et al. (2013), taking into account the form of Eq. (7) coefficients and the assumed values of the system parameters, we can expect that the eigenvalues will be purely real numbers with an absolute value of the order of unity, i.e.: $|s| \sim 1$. In this situation, because the bvp4c solver requires an initial value for the unknown parameter s, it was necessary to carry out preliminary simulations in order to find by the "trial and error" method the region of possible positive eigenvalues. It turns out that such values exist, and therefore the considered system is linearly unstable due to perturbations assumed in the form (6a)–(6b).

For the wavenumber k_y = 100 it is relatively easy to identify the most unstable mode (i.e. the eigenvector $w(z)$ associated with the highest possible eigenvalue s) since the calculations take only few seconds. Once this is achieved, we can start the "continuation method" calculation by taking the eigenvalue found for k_y = 100 as the initial guess in the first step of the loop. We set the variables ky_final = 1e5 and ky_step = 2. For such settings, calculations take several minutes, and intermediate results can be saved to a file and displayed on the screen with a selected frequency (variables: period_print, period_save, and period_plot).

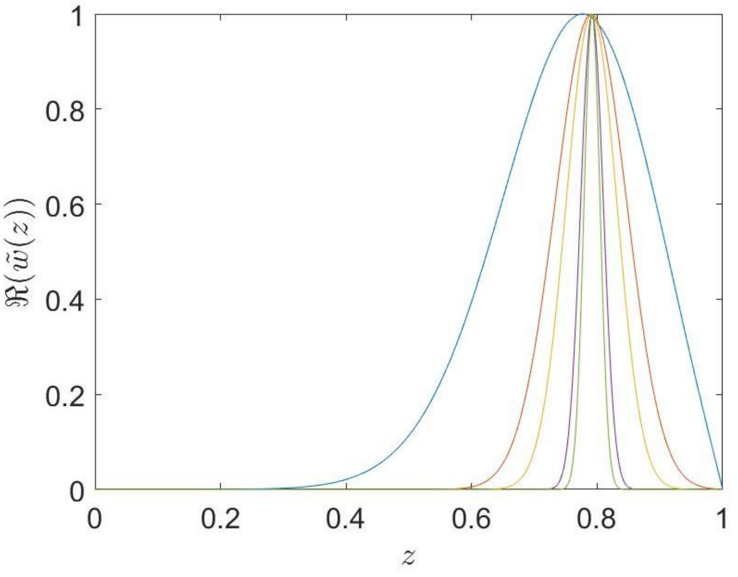

Fig. 3. The most unstable eigenvectors $w(z)$ of Eq. (7) for the values of the parameters (17a)–(17c) and basic state (16a)–(16b), associated with the greatest eigenvalues s from Table 3. Found with the "method of continuation" with the user-defined MATLAB function mainEBVP using the build-in bvp4c solver. Eigenvectors are determined for the following wavenumbers k_y: 100 (blue line), 500 (red line), 1000 (yellow line), 5000 (purple line), and 10000 (green line). The plots are normalized so that max(|w|) = 1.

Table 3 contains eigenvalues s found for several selected wavenumbers k_y and for the other parameters values (17a)–(17c). Figure 3 presents their corresponding eigenvectors $w(z)$, normalized so that $\max(|w|) = 1$. As can be seen on the plot, for increasing wavenumber k_y, the most unstable mode becomes more and more localized around a certain point, which is consistent with the theory (cf. Mizerski et al. 2013).

Table 3
The greatest eigenvalues s of Eq. (7) for selected wavenumbers k_y and for the values of the parameters (17a)–(17c) and basic state (16a)–(16b), found using the "method of continuation" with the user-defined MATLAB function mainEBVP using the build-in bvp4c solver

k_y	100	500	1000	5000	10000
s	0.37437	0.37538	0.37550	0.37560	0.37561

5. SUMMARY

The purpose of this paper was to present the application of the built-in MATLAB bvp4c solver to study the linear stability of magnetohydrodynamic systems. A script using this solver in the form of a user-defined mainEBVP function is discussed and its ready-to-use code is included in the Appendix.

As an example of application, we described the study of the stability of a magnetized electrically conductive fluid layer described by Eqs. (1a)–(1d), due to two-dimensional Fourier disturbances (6a)–(6b) of the basic state given by Eq. (5). We reduced this case to a two-point boundary value problem on the $w(z)$ – the amplitude of vertical component of velocity field perturbation (Eq. (7) with boundary conditions (8)), with the growth rate s determined as the eigenvalue of the linear differential operator associated with this equation.

The script described in Chapter 3 solves the BVP equation (implemented in the form of a system of Eqs. (10a)–(10e)), i.e. for given values of the system parameters it finds eigenvectors and corresponding eigenvalues. All results are displayed in the MATLAB command window, plotted on graphs and saved to files. Chapter 4 discusses exemplary results for both pure real and fully complex eigenvectors and eigenvalues.

From the point of view of linear stability analysis of the considered MHD system, the real case corresponds to the so-called magnetorotational and centrifugal instabilities, and the complex case, additionally, also the instability of magnetic buoyancy. The latter instability analysis was also used as an example of using the bvp4c solver in the so-called "continuation method", i.e. solving the same equation multiple times in a loop, using the solutions as an initial guess in the next calculation step.

The script included in the Appendix can be easily modified and adapted to study the stability of other physical systems. The necessary condition is to reduce the equations describing them to the system of the first order ordinary differential equations of one variable for all unknown functions of physical quantities. These equations do not have to be linear, but they cannot contain more than one derivative of the examined functions in each of them. Often such a representation is possible, so the bvp4c solver can be a useful tool for analysing such systems.

In the case of the eigenvalue problems, such as the one we have presented, it is necessary to take into account the need to initially guess the value of the unknown parameter and the additional boundary condition that follows from this. For this reason, using bvp4c will not be a good idea if the purpose of the analysis is to study the whole spectrum of the differential operator under consideration. However, this solver is perfect for finding eigenvalues close to expected ones or searching for the most unstable eigenvectors, i.e. those associated with the largest eigenvalues. Therefore, the bvp4c is especially useful for verifying the results of theoretical analysis, including those using approximate methods, e.g. perturbation theory, boundary layer method, etc.

Finally, it is worth mentioning that the MATLAB program also has a built-in solver that uses higher-order method, namely the bvp5c. It can easily be used in the script present in this paper. However, our experience shows that calculations using it take much longer and require particularly well-adjusted initial guesses. Also, we observed no essential improvement in the accuracy of the results obtained. This leads to the conclusion that MATLAB bvp4c should be the first choice solver for boundary value problems, and the script included in the Appendix enables its convenient use for a wide range of problems.

Acknowledgments: The author wishes to thank Professor David W. Hughes and Professor Krzysztof A. Mizerski for fruitful discussions on this topic. This work was financed by the National Science Centre, Poland (grant no. 2019/32/C/ST3/00482).

Appendix

THE SOURCE CODE OF THE USER-DEFINED "MAINEBVP" FUNCTION:

```
function mainEBVP()
%% ---------- General information ----------
%
% Function mainEBVP uses MATLAB bvp4c solver to solve the eigenvalue
% problem: the 2nd order linear ODE with non-constant coefficients and with
% given boundary conditions (the boundary value problem):
% W2(z,s)w''(z) + W1(z,s)w'(z) + W0(z,s)w(z) = 0
% w(z=0) = 0
% w(z=1) = 0
% where 's' is the unknown eigenvalue of the equation.
%
% For the eigenvalue problem bvp4c solver requires an additional boundary
% condition to determine the value of s. Hence the main ODE is implemented
% in the nested function odefun as the system of three 1st order ODEs:
% w1'(z) = w2(z)
% w2'(z) = -(W1/W2)w2(z) - (W0/W2)w1(z)
% w3'(z) = w1(z) (or: abs(w1(z))^2, etc.)
% where the last equation for the antiderivative of w1(z) together with two
% additional boundary conditions is the normalization of the solution w1.
% This boundary conditions are implemented in the nested function bcfun as:
% w1(z=0) = 0
% w1(z=1) = 0
% w3(z=0) = 0
% w3(z=1) = 1
%
% Function mainEBVP after simple modifications can potentially solve other
% linear ordinary eigenproblems of higher orders, different BC etc. The
% below version of the mainEBVP function implements as an example the
% equation for the w(z) function, which is the amplitude of the linear
% axisymmetric Fourier-form perturbation of the fluid velocity field in the
% z-direction. The considered fluid is compressible, inviscid, isothermal
% and electrically perfectly conducting, described by the ideal gas law and
% located between the two infinite planes (z=0 and z=1). The fluid in the
% basic equilibrium state is determined by the external magnetic field of
% the form [a0(z), b0, 0], gravity [0,0,-g], rotation [0,R,0] and the shear
% flow [0,u0(z),0]. This basic state is perturbed by the x-independent
% modes varying in the y-direction, which for the velocity field take the
% form: [u(z),v(z),w(z)]*exp(s*t+i*ky*y), where ky is the wavenumber in the
% horizontal y-direction and s is the growth rate of the potential
% instability (complex number in general). It can be shown that the system
% of equations for the perturbations amplitudes can be algebraically
% manipulated to obtain one 2nd order ODE for the function w(z) with growth
% rate s determined as the eigenvalue of the problem. This is the main
% equation solved in this script, and hence the function w1(z) mentioned
% above is the vertical velocity perturbation amplitude w(z) in considered
% physical system. For more details of the similar systems see:
% 1) The case without the rotation and shear flow:
% Mizerski,K.A., Davies,C.R. & Hughes,D.W.(2013),"Short-wavelength magnetic
% buoyancy instability", The Astrophysical Journal Supplement, 205, 16.
% 2) The case from 1) with magnetic and thermal diffusion added:
% Gradzki,M.G. & Mizerski,K.A.(2018),The Effect of Weak Resistivity and
% Weak Thermal Diffusion on Short-wavelengt Magnetic Buoyancy Instability,
% The Astrophysical Journal Supplement, 235, 13
%
% The function mainEBVP optionally uses the so-called "method of
% continuation", where the first solution found is used as a initial guess
% for the next solver calculation, and so forth in the loop. The bvp4c
% solver requires the initial guess for both the solution function and the
% eigenvalue, which can be a problem because the computation takes a long
% time when this input data differs significantly from the output. The
% method of continuation allows to obtain solution even from relatively
% poor initial guesses. What is more, it also allows to change the
% parameter values or the accuracy of calculations in each step of the
% loop, which can be very useful. On the other hand it seems to be less
% effective for the complex functions and eigenvalues. For more details see:
% Jacek Kierzenka(2022), "Tutorial on solving BVPs with BVP4C", MATLAB
% Central File Exchange, Retrieved November 12, 2022.
%
% In the current version, the mainEBVP function takes the following INPUT
% (the numbers correspond to the numbering of the code blocks):
% 01: INPUT: Main parameters [from physical problem]
% 02: INPUT: Basic state functions [from physical problem]
% 03: INPUT: Solver options [see bvp4c MATLAB help]
% 04: INPUT: Initial guess properties [see bvp4c MATLAB help]
% 05: INPUT: OPTIONAL: solving in the loop [for the method of continuation]
%
% The mainEBVP function gives the following OUTPUT:
% 11: OUTPUT: Display results on the screen [MATLAB Command Window]
% 12: OUTPUT: Save data to the files [last solution file & log file]
% 13: OUTPUT: Plots [basic state density; Re, Im, and Abs of the solution]
%
% The mainEBVP function uses the following nested functions:
% w_guess_01  [constant initial guess for the solution]
% w_guess_02  ["half-sine" initial guess for the solution]
% odefun   [main ODE to be solved]
% bcfun       [boundary conditions of the main ODE]
% bcjac       [Jacobians of the boundary conditions]
% extra_space [print numbers with spaces by thousand]
%
% It is worth noting that the mainEBVP function after minor modifications
% can be also used with higher-order MATLAB bvp5c solver.
%
% Abbreviations used in the comments:
% ODE - ordinary  differential equation
% MF  - magnetic field
% SF  - shear flow
% GR  - growth rate of the instability (eigenvalue s of the system)
% BS  - basic (equilibrium) state
% BC  - boundary condition(s)
% IG  - initial guess of the solution
%
% Copyright: M.J.Gradzki 2022©
% Contact: marek.gradzki@gmail.com
```

```
%% ---------- 01:INPUT: Main parameters ----------

% Main nondimensional parameters:
Uu = 1.00;  % shear flow scale
Uo = 0.90;  % rotational frequency scale
P  = 1.50;  % squared "pressure speed" scale
a  = 1.00;  % P*a is the squared isothermal speed of sound scale
L  = 0.25;  % squared Alfven speed scale
f  = 1.00;  % gravity "flag": 1.0="gravity on", 0.0="gravity off"

% Basic state functions parameters:
b0 =  0.01;  % constant poloidal MF (y-direction)
la =  1.4;     % parameter of the basic toroidal MF a0(z)
dz = -1.0;     % parameter of the basic horizontal shear flow u0(z)
R1 =  1.0;     % BC for the basic state density: r0(z=1)=R1

% Horizontal wavenumber (y-direction):
ky = 150;


%% ---------- 02:INPUT: Basic state functions ----------

% Choose one form of the basic shear flow and toroidal magnetic field
% COMMENT the other options!

% % Function handle to ZERO basic toroidal MF a0(z):
% a0_fh = @(z) 0.0;  % MF
% da0   = 0.0;       % MF 1st z-derivative
% dda0  = 0.0;       % MF 2nd z-derivative

% Function handle to LINEAR basic toroidal MF a0(z):
a0_fh = @(z) 1+la.*(1-z);  % MF
da0   = -la;               % MF 1st z-derivative
dda0  = 0.0;               % MF 2nd z-derivative

% % Function handle to ZERO basic SF u0(z):
% u0_fh  = @(z) 0.0;  % SF
% du0_fh = @(z) 0.0;  % SF 1st z-derivative
% ddu0   = 0.0;       % SF 2nd z-derivative

% Function handle to LINEAR basic SF u0(z):
u0_fh  = @(z) dz.*z;  % SF
du0_fh = @(z) dz;     % SF 1st z-derivative
ddu0   = 0.0;         % SF 2nd z-derivative

% % Function handle to QUADRATIC basic shear flow u0(z):
% u0_fh  = @(z) dz.*(z.*z - z + 1);  % SF
% du0_fh = @(z) dz.*(2*z - 1);       % SF 1st z-derivative
% ddu0   = 2*dz;                     % SF 2nd z-derivative


%% ---------- 03:INPUT: Solver options ----------

options = [];  % <- DO NOT CHANGE that line
options = bvpset(options, 'Stats', 'off');        % 'on'/'off'
options = bvpset(options, 'RelTol', 1e-5);        % default: 1e-3
options = bvpset(options, 'AbsTol', 1e-6);        % default: 1e-6
options = bvpset(options, 'Vectorized', 'on');    % 'on'/'off'
options = bvpset(options, 'BCJacobian', @bcjac);  % comment to switch off
options = bvpset(options, 'NMax', 1e20);          % default: floor(1e4/n)


%% ---------- 04:INPUT: Initial guess properities ----------

% Initial value of the growth rate s (eigenvalue):
sRe =  0.58;  % real part
```

```
sIm = -0.28;  % imaginary part

% Properities of the initial guess (IG) of the solution:
za = 0.0;  % left boundary of the domain
zb = 1.0;  % right boundary of the domain
meshPointsNumber = 1+2e1;  % number of points of the IG mesh
solinitNumber    = '01';   % IG solution type: '01'=constant, '02'=sine


%% ---------- 05:INPUT: Solving in the loop (OPTIONAL)  ----------
% Solving 'by continuation' in a loop.
% Uses previously obtained solution as initial guess it next step
% Possible change of parameter values in subsequent steps (ky, RelTol,...),
% to obtaine high values of ky or better tolerance

% Loop Calculation Switch:
loop_switch = 'off';  % 'on'/'off'

% Choose loop with ky changing ('on') or constant ('off'):
loop_over_ky = 'off';  % 'on'/'off'

% Set for the loop with changing ky:
ky_final = ky + 10;  % the greatest ky in the loop / ky + 1000.0
ky_step  = 1.0;        % ky increment in the loop

% Set for the loop with constant ky:
loop_steps_number = 10;

% Choose the frequency of printing, saving and ploting:
period_print = 1;  % Period of printing on the screen [unit=step]
period_save  = 1;  % Period of saving to the file [unit=step]
period_plot  = 1;  % Period of drawing the plot [unit=step]

% Plot delay:
pause_time_plot = 0.0;  % 0.0=no pause in drawing new plots in the loop


%% ---------- 06:SOLVER: Creation of the initial guess ----------

% Initial value of the growth rate s (eigenvalue):
s_init  = sRe + 1i*sIm;

% Uniform mesh for initial guess:
z_mesh = linspace(za, zb, meshPointsNumber);  % (0, 1e<n>, 1+1e<n>)

% Create initial guess of the solution using solinit:
if strcmp(solinitNumber, '01')
    solinit = bvpinit(z_mesh, @w_guess_01, s_init);
elseif strcmp(solinitNumber, '02')
    solinit = bvpinit(z_mesh, @w_guess_02, s_init);
end


%% ---------- 07:SOLVER: Calculation of the solution ----------

% Convenient definition: 'P' and 'a' appear only in the product:
Pa = P*a;

% Basic state equation for the basic density r0(z):
myode = @(z,r) ((Uu*Uo*u0_fh(z)-f)/Pa)*r - (L/Pa)*a0_fh(z)*da0;

% Solution of the basic state equation for the basic density r0(z):
sol_r0 = ode45(myode, [1 0], R1);  % BC: r0(z=1)=R1

% Evaluation of the basic density r0(z) (used by nested functions):
r0_fh = @(z) deval(sol_r0, z, 1);  % '1' is the first part of sol vector
```

```
tic  % start stopwatch timer
start_date = datestr(now);  % start date and time of the calculation
fprintf('%-5s\n', ['Start: ', start_date]);  % print start date and time

% Print initial setup info:
ky_init = ky;
fprintf('ky   = %.1f\n', ky_init)
fprintf('b0   = %.1e\n', b0)
fprintf('Init sigma = %+.15f %+.15fi \n', sRe, sIm)
fprintf(['Init guess = ', solinitNumber, ' (number) \n'])
fprintf('Init mesh  = %1.0f points \n', meshPointsNumber)
fprintf('RelTol = %.1e\n', bvpget(options,'RelTol',1e-3))
fprintf('AbsTol = %.1e\n', bvpget(options,'AbsTol',1e-6))

% !!! The first and main use of the solver...
% ... and the only one, if loop_switch = 'off'
sol = bvp4c(@odefun, @bcfun, solinit, options);

% Components of the variable sol (created by bvp4c):
% sol.x = final mesh of the domain z=[0,1]
% sol.y(1,:) = w1(x) = w(z) = amplitude of z-comp. of velocity perturb.
% sol.y(2,:) = w2(x) = w'(z) = derivative of w(z)
% sol.y(3,:) = w3(x) = function used for normalization of w(z)

% Stop the timer and set the date if loop_switch = 'off'
if strcmp(loop_switch, 'off')
    time = round(toc);
    end_date = datestr(now);
end

loop_steps = 1; % DO NOT CHANGE whether the loop_switch is 'on' or 'off'


%% ---------- 08:SOLVER: OPTIONAL: Solving in a loop ----------
% Solving 'by continuation' in a loop (see ).
% Uses previously obtained solution as initial guess it next step
% Possible change of parameter values in subsequent steps (ky, RelTol,...),
% to obtaine high values of ky or better tolerance

if strcmp(loop_switch, 'on')

    % Open the file to save loop-solving results:
    loop_sol = fopen('loop_sol.txt','w');

    % Save initial setup info to the file:
    fprintf(loop_sol, 'ky   = %.1f\n', ky_init);
    fprintf(loop_sol, 'b0   = %.1e\n', b0);
    fprintf(loop_sol, 'Init sigma = %+.15f %+.15fi \n', sRe, sIm);
    fprintf(loop_sol, ['Init guess = ', solinitNumber, ' (number) \n']);
    fprintf(loop_sol, 'Init mesh  = %1.0f points \n', meshPointsNumber);

    % Print & save info about the first solution found in the section 07:
    [~, x_max_ind] = max(abs(sol.y(1,:)));
    x_plot = sol.x(x_max_ind);  % argument of the maximum of abs(sol.y(1))
    RelTolLoop = bvpget(options,'RelTol');
    AbsTolLoop = bvpget(options,'AbsTol');
    % Print on the screen:
    fprintf('For ky = %5.6e, s = %+4.15f %+4.15fi, z_loc = %+4.15f,',...
        ky, real(sol.parameters), imag(sol.parameters), x_plot);
    fprintf(' RelTol = %5.1e, AbsTol = %5.1e. Step %1.d \n',...
        RelTolLoop, AbsTolLoop, 1);
    % Save in the file:
    fprintf(loop_sol, 'For ky = %5.6e, s = %+4.15f %+4.15fi, z_loc = %+4.15f,',...
        ky, real(sol.parameters), imag(sol.parameters), x_plot);
    fprintf(loop_sol, ' RelTol = %5.1e, AbsTol = %5.1e. Step %1.d \n',...
        RelTolLoop, AbsTolLoop, 1);
    % Creation of the main PLOT of this section:
    figLoop = figure('Name','Solving in the loop');
    hold on
    pause_time = pause_time_plot;

    % Plot of the REAL part of the initial guess of w1(z):
    plot(solinit.x, real(solinit.y(1,:))/max(abs(solinit.y(1,:))), '-');
    title('Solving in the loop', 'interpreter','latex');
    xlabel('$$z$$', 'interpreter','latex')
    ylabel('$$\Re[w(z)]$$', 'interpreter','latex')
    set(gca, 'FontSize', 15)
    xlim([0.0 1.0])
    % ylim([0.0 1.0])
    % grid on
    drawnow
    pause(pause_time)

    % Plot normalized first solution found in previous section:
    plot(sol.x, real(sol.y(1,:))/max(abs(sol.y(1,:))), '-');
    drawnow
    pause(pause_time)

    % Loop setup:
    k_init  = ky;  % Do NOT change!!!
    % If loop with changing ky was chosen in section 05 we have:
    if strcmp(loop_over_ky, 'on')
        loop_steps_number = 1 + ceil((ky_final-k_init)/ky_step);
    end

    % In each step solve and then optionally print, plot, save to file:
    for i1 = 2:loop_steps_number
        if strcmp(loop_over_ky, 'on')
            ky = ky + ky_step;  % for loop with changing ky
        end
        % % Uncomment ONLY for loop over the RelTol:
        % RelTolLoop = RelTolLoop/10;
        % options = bvpset(options, 'RelTol', RelTolLoop);
        % % Uncomment ONLY for loop over the AbsTol:
        % AbsTolLoop = AbsTolLoop/10;
        % options = bvpset(options, 'AbsTol', AbsTolLoop);
        % Solve in the current loop step:
        sol = bvp4c(@odefun, @bcfun, sol, options);
        % Find argument z of the abs(w1(z)) maximum:
        [y_max, x_max_ind] = max(abs(sol.y(1,:)));
        x_plot = sol.x(x_max_ind);
        % OPTIONALLY print, plot, save to file with period chosen in sec.05
        if mod(i1, period_print) == 0  % Print on the screen:
            fprintf('For ky = %5.6e, s = %+4.15f %+4.15fi, z_loc = %+4.15f,',...
                ky, real(sol.parameters), imag(sol.parameters), x_plot);
            fprintf(' RelTol = %5.1e, AbsTol = %5.1e. Step %1.d \n',...
                RelTolLoop, AbsTolLoop, i1);
        end
        if mod(i1, period_save) == 0  % Save to file:
            fprintf(loop_sol, 'For ky = %5.6e, s = %+4.15f %+4.15fi, z_loc =
%+4.15f,',...
                ky, real(sol.parameters), imag(sol.parameters), x_plot);
            fprintf(loop_sol, ' RelTol = %5.1e, AbsTol = %5.1e. Step %1.d \n',...
                RelTolLoop, AbsTolLoop, i1);
        end
        if mod(i1, period_plot) == 0  % Plot Re(w1(z)) on the main plot:
            plot(sol.x, real(sol.y(1,:))/y_max, '-');
            drawnow
            pause(pause_time)
        end
        loop_steps = loop_steps + 1; % ONLY for display & strings!!!
    end
    hold off
    % Close file with saved loop results:
    fprintf(loop_sol, '%-5s\n', '');  % adds empty line
    fclose(loop_sol);

    % If the loop_switch = 'on':
    time = round(toc);  % The timer stops
    end_date = datestr(now);  % End date stops

    % plot to file:
    saveas(figLoop,'figLoop')  % plot to .fig file
    orient(figLoop,'landscape')
    print('figLoop','-dpdf','-fillpage') % plot to .pdf file
    orient(figLoop,'portrait')
    print('figLoop','-djpeg') % plot to .jpeg file
end


%% ---------- 09:SOLVER: Additional calculations ----------

% Final ky and s (for strings in plots and files):
ky = ky(end);  % It works fine even if ky is a scalar
s_fin = sol.parameters;

% index of argument z for max value of w1(z):
[~, x_max_ind] = max(abs(sol.y(1,:)));
% argument of this max value:
x_fin = sol.x(x_max_ind);

% Some integrals of the solution w1(z):
S_w1      = trapz(sol.x, sol.y(1,:));          % Int. of Re(w1)+i*Im(w1)
S_w1_abs  = trapz(sol.x, abs(sol.y(1,:)));     % Int. of Abs(w1)
S_w1_sq   = trapz(sol.x, sol.y(1,:).^2);       % Int. of Re(w1^2)+i*Im(w1^2)
S_w1_abs2 = trapz(sol.x, abs(sol.y(1,:)).^2); % Int. of Abs(w1)^2
% These integrals are used to check the normalization condition ...
... namely the last EQ in odefun(z,w,s)


%% ---------- 10:SOLVER: Creating strings for plots and files ----------

% Proper time unit (to dispaly) is automatically selected here:
if (60<=time) % && (time<=3600)
    time = (time/60);
    calcTimeUnit = char(' min');
elseif time>3600
    time = (time/3600);
    calcTimeUnit = char(' h');
else
    calcTimeUnit = char(' s');
end

% Proper f-jacobian info (to dispaly) is automatically selected here:
if isempty(bvpget(options,'FJacobian'))
    FJac = 'FJac.  = off';
else
    FJac = 'FJac.  = on';
end

% Proper BC-jacobian info (to dispaly) is automatically selected here:
if isempty(bvpget(options,'BCJacobian'))
    BCJac = 'BCJac. = off';
else
    BCJac = 'BCJac. = on';
end

% Strings for plot annotation and for data files:
str01 = { ...
```

```
      ['Start: ', start_date] ...
    , ['End  : ' end_date] ...
    , ['b0   = ', num2str(b0,'%+.1e\n')] ...
    , ['ky   = ', extra_space(ky_init)] ...
    , ['ky FINAL         = ', extra_space(ky), ' ~ ', num2str(ky,'%.0e')] ...
    , ['sigma FINAL      = ', num2str([real(s_fin),imag(s_fin)],'%+.15f 
%+.15fi\n')] ...
    , ['sigma init.guess = ', num2str([sRe, sIm],'%+.15f %+.15fi\n')] ...
    , ['z.max FINAL      = ', num2str([real(x_fin),imag(x_fin)],'%+.15f %+.2fi\n')] 
...
    , ['Integral  w1     = ', num2str([real(S_w1),imag(S_w1)],'%+.8f %+.8fi\n')] 
...
    , ['Integral |w1|    = ', num2str(S_w1_abs,'%+.8f\n')] ...
    , ['Integral  w1*w1  = ', num2str([real(S_w1_sq),imag(S_w1_sq)],'%+.8f 
%+.8fi\n')] ...
    , ['Integral |w1*w1| = ', num2str(S_w1_abs2,'%+.8f\n')] ...
    , ['Total.calculation.time    = ', num2str(time, '%10.1f\n'),calcTimeUnit] ...
    , ['Loop steps number         = ', num2str(loop_steps,'%10.d\n')] ...
    , ['Solution init. guess no.  = ', num2str(solinitNumber,'%10.d\n')] ...
    , ['Initial mesh pts.number   = ', num2str(numel(z_mesh),'%10.d\n')] ...
    , ['Final mesh points number  = ', num2str(sol.stats.nmeshpoints,'%10.d\n')] 
...
    , ['Final ODEfun evals.number = ', num2str(sol.stats.nODEevals,'%10.d\n')] ...
    , ['Final BCfun  evals.number = ', num2str(sol.stats.nBCevals,'%10.d\n')] ...
    , ['Final.maximum.residual    = ', num2str(sol.stats.maxres,'%10.1e\n')] ...
    , ['Solver = bvp4c'] ...
    , ['RelTol = ', num2str(bvpget(options,'RelTol',1e-3),'%10.1e\n')] ...
    , ['AbsTol = ', num2str(bvpget(options,'AbsTol',1e-6),'%10.1e\n')] ...
    , ['NMax   = ', num2str(bvpget(options,'NMax','def.'),'%10.1e\n')]...
    , ['Vect.  = ', bvpget(options,'Vectorized'),]...
    , FJac ...
    , BCJac
    };


%% ---------- 11:OUTPUT: Display results on the screen ----------

% Print result at the command window:
fprintf('---------------- OUTPUT START ----------------\n');
for i2 = 1:numel(str01)
        fprintf('%-5s\n', str01{1,i2});
end
fprintf('------------------ OUTPUT END ------------------\n');
fprintf(' \n');


%% ---------- 12:OUTPUT: Save data to the files ----------

% Save calculation details to the file (without overwriting):
f00_info = fopen('mainEBVP_log.txt','a');
fprintf(f00_info, '---------------- LOG START ----------------\n');
for i3 = 1:numel(str01)
    fprintf(f00_info, '%-5s\n', str01{1,i3});
end
fprintf(f00_info, '---------------- LOG END ------------------\n');
fprintf(f00_info, '\n');
fprintf(f00_info, '\n');
% fprintf(f00_info, '%-5s\n', '');
fclose(f00_info);


% Save NONnormalized solution sol to the file (with overwriting):
f01_sol = fopen('mainEBVP_last_sol.txt','w');
for i4 = 1:numel(str01)
    fprintf(f01_sol, '%-5s\n', str01{1,i4}); % Save calculation details
end
fprintf(f01_sol, '%-5s\n', '');
fprintf(f01_sol, '%+17s %+17s %+17s %+17s\n',...
    'sol.x', 'sol.y1', 'sol.y2', 'sol.y3');
fprintf(f01_sol, '%+12.10e %+12.10e %+12.10e %+12.10e\n', ...
    [sol.x; sol.y(1,:); sol.y(2,:); sol.y(3,:)]); % Save the solution
fclose(f01_sol);


%% ---------- 13:OUTPUT: Plots ----------

% Plot of the basic state density r0(z), calculated in the section 07:
fig0r0 = figure('Name','Basic state density r0(z)');
plot(sol_r0.x, sol_r0.y, '-', 'MarkerSize',10)
title('Basic state density $$\rho_{0}(z)$$', 'interpreter','latex');
xlabel('$$z$$', 'interpreter','latex')
ylabel('$$\rho_{0}(z)$$', 'interpreter','latex')
set(gca, 'FontSize', 15)
xlim([0.0 1.0])
% Save plot to files:
saveas(fig0r0,'fig_BasicStateDensity')  % plot to .fig file
orient(fig0r0,'landscape')
print('fig_BasicStateDensity','-dpdf','-fillpage')  % plot to .pdf file
orient(fig0r0,'portrait')
print('fig_BasicStateDensity','-djpeg')  % plot to .jpeg file


% Plot of the normalized real part of the solution w1(z), i.e.: Re[w(z)]:
fig01real = figure('Name','Real part of the normalized solution: Re[w(z)]');
plot(sol.x, real(sol.y(1,:))/max(abs(real(sol.y(1,:)))), '-', ...
     'MarkerSize', 10)
title('Real part of the normalized solution: $$\Re[w(z)]$$', 'interpreter','la-
tex');
xlabel('$$z$$', 'interpreter','latex')
ylabel('$$\Re[w(z)]$$', 'interpreter','latex')
set(gca, 'FontSize', 15)
xlim([0.0 1.0])
% Save plot to files:
saveas(fig01real,'fig_solution_real_y1')  % plot to .fig file
orient(fig01real,'landscape')
print('fig_solution_real_y1','-dpdf','-fillpage')  % plot to .pdf file
orient(fig01real,'portrait')
print('fig_solution_real_y1','-djpeg')  % plot to .jpeg file


% Plot of the normalized imaginary part of the solution w1(z), i.e.: Im[w(z)]:
fig01imag = figure('Name','Imaginary part of the normalized solution: Im[w(z)]');
plot(sol.x, imag(sol.y(1,:))/max(abs(imag(sol.y(1,:)))), '-', ...
     'MarkerSize', 10)
title('Imaginary part of the normalized solution: $$\Im[w(z)]$$', 'interpret-
er','latex');
xlabel('$$z$$', 'interpreter','latex')
ylabel('$$\Im[w(z)]$$', 'interpreter','latex')
set(gca, 'FontSize', 15)
xlim([0.0 1.0])
% Save plot to files:
saveas(fig01imag,'fig_solution_imag_y1')  % plot to .fig file
orient(fig01imag,'landscape')
print('fig_solution_imag_y1','-dpdf','-fillpage')  % plot to .pdf file
orient(fig01imag,'portrait')
print('fig_solution_imag_y1','-djpeg')  % plot to .jpeg file


% Plot of the normalized modulus of the solution w1(z), i.e.: |w(z)|:
fig01abs = figure('Name','Moduls of the normalized solution: |w(z)|');
plot(sol.x, abs(sol.y(1,:))/max(abs(sol.y(1,:))), '-', ...
     'MarkerSize', 10)
title('Moduls of the normalized solution: $$|w(z)|$$', 'interpreter','latex');
xlabel('$$z$$', 'interpreter','latex')
```

```
ylabel('$$|w(z)|$$', 'interpreter','latex')
set(gca, 'FontSize', 15)
xlim([0.0 1.0])
% Save plot to files:
saveas(fig01abs,'fig_solution_abs_y1')  % plot to .fig file
orient(fig01abs,'landscape')
print('fig_solution_abs_y1','-dpdf','-fillpage')  % plot to .pdf file
orient(fig01abs,'portrait')
print('fig_solution_abs_y1','-djpeg')  % plot to .jpeg file


% Plot of all normalized parts of the solution w1(z), i.e.:
% Re[w(z)], Im[w(z)] and Abs[w(z)]:
fig01all = figure('Name','All parts of the normalized solution w(z)');
plot(sol.x, real(sol.y(1,:))/max(abs(sol.y(1,:))), '-', ...
     sol.x, imag(sol.y(1,:))/max(abs(sol.y(1,:))),  '-', ...
     sol.x,  abs(sol.y(1,:))/max(abs(sol.y(1,:))),  '-', ...
     'MarkerSize', 10)
title('All parts of the normalized solution: $$w(z)$$', 'interpreter','latex');
xlabel('$$z$$', 'interpreter','latex')
ylabel('$$w(z)$$', 'interpreter','latex')
leg1 = legend('$$\Re[w(z)]$$', '$$\Im[w(z)]$$', '$$|w(z)|$$');
legend('Location','northwest')
set(leg1,'Interpreter','latex');
set(gca, 'FontSize', 15)
xlim([0.0 1.0])
% Save plot to files:
saveas(fig01all,'fig_solution_all_y1')  % plot to .fig file
orient(fig01all,'landscape')
print('fig_solution_all_y1','-dpdf','-fillpage')  % plot to .pdf file
orient(fig01all,'portrait')
print('fig_solution_all_y1','-djpeg')  % plot to .jpeg file


%% ---------- Nested functions ----------

% Components of the variable sol (created by bvp4c):
% sol.x = final mesh of the domain z=[0,1]
% sol.y(1,:) = w1(z) = w(z) = amplitude of z-comp. of velocity perturb.
% sol.y(2,:) = w2(z) = w'(z) = derivative of w(z)
% sol.y(3,:) = w3(z) = function used for normalization of w(z)

% ----- Initial guess 01: constant -----
    function w_init = w_guess_01(z)
    w_init = [1; 1; 1];
    end  % w_guess_01


% ----- Initial guess 02: sine-type -----
    function w_init = w_guess_02(z)
    w_init = [sin(1*pi*z)
         pi*cos(1*pi*z)
         -(1/pi)*cos(1*pi*z)];
    end  % w_guess_02


% ----- Main ODE to be solved -----
    function dwdz = odefun(z,w,s)
    % All formulae in this function are VECTORIZED!
    % WORKS OK even for 'Vectorization' = 'off' !!!!
     r0 =  r0_fh(z);  % r0(z): basic state density calc. from BS ODE
     a0 =  a0_fh(z);  % a0(z): toroidal MF
     u0 =  u0_fh(z);  % u0(z): horizontal shear flow
    du0 = du0_fh(z);  % u0'(z): z-derivative of the horizontal SF
    % Relations obtained from basic state equation:
    dr0  = ((Uu*Uo*u0-f)/Pa).*r0 - (L/Pa)*da0.*a0;
    ddr0 = ((Uu*Uo*u0-f)/Pa).*dr0 + (Uu*Uo/Pa).*r0.*du0 ...
```

```
       - (L/Pa).*((da0).^2 + a0.*dda0);
% Main coefficients of the 2nd order ODE for w(z):
W2 = (-b0.^2.*ky.^2.*L - r0.*s.^2).* ...
    ((a0.^2.*L + Pa.*r0).*s.^2 + b0.^2.*L.*(ky.^2.*Pa + s.^2)).* ...
    (b0.^2.*ky.^2.*L.*(ky.^2.*Pa + s.^2) + ...
    s.^2.*(a0.^2.*ky.^2.*L + r0.*(ky.^2.*Pa + s.^2)));
W1 = (-b0.^4).*ky.^6.*L.^2.*Pa.*s.^2.*(a0.*da0.*L + 2.*dr0.*Pa +...
    r0.*(f - u0.*Uo.*Uu)) - r0.^2.*s.^8.*(3.*a0.*da0.*L + 2.*dr0.*Pa +...
    r0.*(f - u0.*Uo.*Uu)) + ky.^2.*((-(a0.^2.*L + Pa.*r0)).*s.^6.*...
    (a0.^2.*dr0.*L + a0.*da0.*L.*r0 + r0.*(2.*dr0.*Pa + r0.*(f - ...
    u0.*Uo.*Uu))) - b0.^2.*L.*s.^6.*(a0.^2.*dr0.*L + 4.*a0.*da0.*L.*r0 +...
    2.*r0.*(2.*dr0.*Pa + r0.*(f - u0.*Uo.*Uu)))) + ky.^4.*((-b0.^4).*...
    L.^2.*s.^4.*(a0.*da0.*L + 2.*dr0.*Pa + r0.*(f - u0.*Uo.*Uu)) -...
    b0.^2.*L.*s.^4.*(a0.^3.*da0.*L.^2 + 2.*a0.*da0.*L.*Pa.*r0 +...
    2.*Pa.*r0.*(2.*dr0.*Pa + r0.*(f - u0.*Uo.*Uu)) + a0.^2.*L.*...
    (3.*dr0.*Pa + r0.*(f - u0.*Uo.*Uu))));
W0 = b0.^6.*ky.^10.*L.^3.*Pa.^2 - 1i.*b0.*da0.*ky.*L.*r0.^2.*s.^7.*Uo +...
    r0.^2.*s.^8.*((-dr0).*f - da0.^2.*L - a0.*dda0.*L - ddr0.*Pa + ...
    dr0.*u0.*Uo.*Uu + r0.*(s.^2 + Uo.^2 + du0.*Uo.*Uu)) - ...
    1i.*a0.*b0.^3.*ky.^7.*L.^2.*Pa.*s.*Uo.*(a0.*da0.*L + dr0.*Pa + ...
    r0.*(-f + u0.*Uo.*Uu)) + ky.^8.*(2.*b0.^6.*L.^3.*Pa.*s.^2 + ...
    b0.^4.*L.^2.*Pa.*(2.*a0.^2.*L.*s.^2 + Pa.*r0.*(3.*s.^2 + du0.*Uo.*Uu)...
    + a0.*da0.*L.*(f - u0.*Uo.*Uu))) + ky.^6.*(b0.^6.*L.^3.*s.^4 + ...
    b0.^4.*L.^2.*s.^2.*(2.*a0.^2.*L.*s.^2 - Pa.*(dr0.*f + da0.^2.*L + ...
    ddr0.*Pa - dr0.*u0.*Uo.*Uu - 2.*r0.*(3.*s.^2 + du0.*Uo.*Uu)) + ...
    a0.*L.*((-dda0).*Pa + da0.*(f - u0.*Uo.*Uu))) + b0.^2.*L.* ...
    s.^2.*(a0.^4.*L.^2.*s.^2 + Pa.^2.*r0.^2.*(3.*s.^2 + Uo.^2 + ...
    2.*du0.*Uo.*Uu) + a0.^3.*da0.*L.^2.*(f - u0.*Uo.*Uu) + ...
    2.*a0.*da0.*L.*Pa.*r0.*(f - u0.*Uo.*Uu) + a0.^2.*L.*Pa.*(r0.*(4.*s.^2 +...
    Uo.^2 + 2.*du0.*Uo.*Uu) + dr0.*(-f + u0.*Uo.*Uu)))) + ...
    ky.^3.*((-1i).*b0.^3.*L.^2.*(a0.*dr0 + da0.*r0).*s.^5.*Uo - ...
    1i.*b0.*L.*s.^5.*Uo.*(a0.^3.*dr0.*L + da0.*Pa.*r0.^2 + ...
    a0.*r0.*(dr0.*Pa + r0.*(-f + u0.*Uo.*Uu)))) + ...
    ky.^5.*((-1i).*a0.*b0.*L.*(a0.^2.*L + Pa.*r0).*s.^3.*Uo.*(a0.*da0.*L +...
    dr0.*Pa + r0.*(-f + u0.*Uo.*Uu)) - ...
    1i.*b0.^3.*L.^2.*s.^3.*Uo.*(a0.^2.*da0.*L + da0.*Pa.*r0 + ...
    a0.*(2.*dr0.*Pa + r0.*(-f + u0.*Uo.*Uu)))) + ...
    ky.^4.*((-b0.^4).*L.^2.*s.^4.*(dr0.*f + da0.^2.*L + a0.*dda0.*L + ...
    ddr0.*Pa - dr0.*u0.*Uo.*Uu - r0.*(3.*s.^2 + du0.*Uo.*Uu)) + ...
    (a0.^2.*L + Pa.*r0).*s.^4.*(Pa.*r0.^2.*(s.^2 + Uo.^2 + du0.*Uo.*Uu) +...
    a0.*da0.*L.*r0.*(f - u0.*Uo.*Uu) + a0.^2.*L.*(r0.*(s.^2 + Uo.^2 + ...
    du0.*Uo.*Uu) + dr0.*(-f + u0.*Uo.*Uu))) + ...
    b0.^2.*L.*s.^4.*((-a0.^3).*dda0.*L.^2 + 2.*Pa.*r0.*((-dr0).*f - ...
    da0.^2.*L - ddr0.*Pa + dr0.*u0.*Uo.*Uu + r0.*(3.*s.^2 + Uo.^2 + ...
    2.*du0.*Uo.*Uu)) + a0.^2.*L.*(-2.*dr0.*f + da0.^2.*L - ddr0.*Pa + ...
    2.*dr0.*u0.*Uo.*Uu + r0.*(4.*s.^2 + Uo.^2 + 2.*du0.*Uo.*Uu)) + ...
    2.*a0.*L.*(da0.*dr0.*Pa + r0.*((-dda0).*Pa + da0.*(f - u0.*Uo.*Uu))))) +...
    ky.^2.*(b0.^2.*L.*r0.*s.^6.*(r0.*(3.*s.^2 + Uo.^2 + 2.*du0.*Uo.*Uu) -...
    2.*(da0.^2.*L + a0.*dda0.*L + ddr0.*Pa + dr0.*(f - u0.*Uo.*Uu))) +...
    s.^6.*((-a0.^3).*L.^2.*(da0.*dr0 + dda0.*r0) + Pa.*r0.^2.*((-dr0).*f -...
    da0.^2.*L - ddr0.*Pa + dr0.*u0.*Uo.*Uu + 2.*r0.*(s.^2 + Uo.^2 + ...
    du0.*Uo.*Uu)) + a0.*L.*r0.*(2.*da0.*dr0.*Pa + r0.*((-dda0).*Pa + ...
    da0.*(f - u0.*Uo.*Uu))) + a0.^2.*L.*((-dr0.^2).*Pa + 2.*r0.^2.*(s.^2 +...
    Uo.^2 + du0.*Uo.*Uu) + r0.*(da0.^2.*L - ddr0.*Pa - ...
    2.*dr0.*(f - u0.*Uo.*Uu)))));
% Structure of the main system of equations:
% dw1/dz = w2
% dw2/dz = [main eq]
% dw3/dz = abs(w1)^2  % exemplary normalization of w1(z) function
dwdz = [w(2,:)
        -(W1./W2).*w(2,:)-(W0./W2).*w(1,:)
        abs(w(1,:)).^2];
        % Normalization options for the last equation:
        % w(1,:) / abs(w(1,:)) / w(1,:).^2 / abs(w(1,:)).^2
        % First one is the fastest, last one is the saftiest!
end  % odefun
```

```
% ----- Boundary conditions -----
    function res = bcfun(wa,wb,s)
    % Three BC + additional 4th BC to determine the eigenvalue!
    res = [wa(1)
           wb(1)
           wa(3)       % normalization of w1
           wb(3)-(1)]; % normalization of w1
    end  % bcfun


% ----- Jacobian of the boundary conditions -----
    function [dbcdwa,dbcdwb,dbcds] = bcjac(z,w,s)
    % Analytical partial derivatives of bcfun(wa,wb,s):
    dbcdwa = [1 0 0
              0 0 0
              0 0 1
              0 0 0];
    dbcdwb = [0 0 0
              1 0 0
              0 0 0
              0 0 1];
    dbcds  = [0; 0; 0; 0];
    end  % bcjac


% ----- Spaces in chars-numbers -----
    function numOut = extra_space(numIn)
    % Function gives char or string with spaces by thousand, e.g.:
    % e.g.: extra_space(1e7) = '10 000 000'
    jf = java.text.DecimalFormat;
    numOut = char(jf.format(numIn));  % omit "char" if you want a string
    end  % extra_space


end  % MAIN: mainEBVP
```

References

Acheson, D.J. (1979), Instability by magnetic buoyancy, *Solar Phys.* **62**, 23–50, DOI: 10.1007/BF00150129.

Ackleh, A.S., E.J. Allen, R.B. Kearfott, and P. Seshaiyer (2010), *Classical and Modern Numerical Analysis*, Chapman and Hall/CRC, New York, DOI: 10.1201/b12332.

Arnol'd, V.I. (1992), *Ordinary Differential Equations*, Springer, Berlin, Heidelberg.

Balbus, S.A., and J.F. Hawley (1991), A powerful local shear instability in weakly magnetized disks. I. Linear analysis, *Astrophys. J.* **376**, 214, DOI: 10.1086/170270.

Batchelor, G.K. (2000), *An Introduction to Fluid Dynamics*, Cambridge University Press, Cambridge, DOI: 10.1017/CBO9780511800955.

Chandrasekhar, S. (1961), *Hydrodynamic and Hydromagnetic Stability*, Clarendon Press, Oxford.

Coleman, M.P. (2013), *An Introduction to Partial Differential Equations with MATLAB*, 2nd ed., Chapman and Hall/CRC, New York, DOI: 10.1201/b15058.

Drazin, P.G., and W.H. Reid (2004), *Hydrodynamic Stability*, 2nd ed., Cambridge University Press, Cambridge, DOI: 10.1017/CBO9780511616938.

Gilman, P.A. (1970), Instability of magnetohydrostatic stellar interiors from magnetic buoyancy. I., *Astrophys. J.* **162**, 1019, DOI: 10.1086/150733.

Gilman, P.A. (2018a), Magnetic buoyancy and magnetorotational instabilities in stellar tachoclines for solar- and antisolar-type differential rotation, *Astrophys. J.* **867**, 1, 45, DOI: 10.3847/1538-4357/aae08e.

Gilman, P.A. (2018b), Magnetic buoyancy and rotational instabilities in the tachocline, *Astrophys. J.* **853**, 1, 65, DOI: 10.3847/1538-4357/aaa4f4.

Gilman, P., and M. Dikpati (2014), Baroclinic instability in the solar tachocline, *Astrophys. J.* **787**, 1, 60, DOI: 10.1088/0004-637X/787/1/60.

Glendinning, P. (1994), *Stability, Instability and Chaos: An Introduction to the Theory of Nonlinear Differential Equations*, Cambridge University Press, Cambridge, DOI: 10.1017/CBO9780511626296.

Gradzki, M.G., and K.A. Mizerski (2018), The effect of weak resistivity and weak thermal diffusion on short-wavelength magnetic buoyancy instability, *The Astrophys. J. Suppl. Ser.* **235**, 13, DOI: 10.3847/1538-4365/aaa408.

Hughes, D.W. (1985), Magnetic buoyancy instabilities incorporating rotation, *Geophys. Astrophys. Fluid Dynam.* **34**, 1–4, 99–142, DOI: 10.1080/03091928508245440.

Keskin, A.Ü. (2019), *Boundary Value Problems for Engineers with MATLAB Solutions*, Springer, Cham, DOI: 10.1007/978-3-030-21080-9.

Kierzenka, J. (2022), Tutorial on solving BVPs with BVP4C, *MATLAB Central File Exchange*, available from: https://www.mathworks.com/matlabcentral/fileexchange/3819-tutorial-on-solving-bvps-with-bvp4c.

Lamb, H. (1975), *Hydrodynamics*, 6th ed., Cambridge University Press, Cambridge.

Lanczos, C. (1996), *Linear Differential Operators*, Classics in Applied Mathematics Ser., Vol. 18, Society for Industrial and Applied Mathematics, Philadelphia, DOI: 10.1137/1.9781611971187.

Mizerski, K.A., C.R. Davies, and D.W. Hughes (2013), Short-wavelength magnetic buoyancy instability, *Astrophys. J. Suppl. Ser.* **205**, 2, 16, DOI: 10.1088/0067-0049/205/2/16.

Moffat, H.K. (1978), *Magnetic Field Generation in Electrically Conducting Fluids*, Cambridge University Press, Cambridge.

Nemyckij, V.V., and V.V. Stepanov (1989), *Qualitative Theory of Differential Equations*, Dover Publications, New York.

Pedlosky, J. (1987), *Geophysical Fluid Dynamics*, 2nd ed., Springer, New York.

Roberts, P.H. (1967), *An Introduction to Magnetohydrodynamics*, Longmans, London.

Shampine, L.F., I. Gladwell, and S. Thompson (2003), *Solving ODEs with MATLAB*, Cambridge University Press, Cambridge, DOI: 10.1017/CBO9780511615542.

Shampine, L.F., and J. Kierzenka (2001), A BVP solver based on residual control and the MATLAB PSE, *ACM Trans. Math. Softw.* **27**, 3, 299–316, DOI: 10.1145/502800.502801.

Smith, G.D. (1985), *Numerical Solution of Partial Differential Equations: Finite Difference Methods*, 3rd ed., Oxford University Press, Oxford.

Trefethen, L.N. (2000), *Spectral Methods in MATLAB*, Oxford University Press, Oxford, DOI: 10.1137/1.9780898719598.

Received 3 December 2022
Received in revised form 30 December 2022
Accepted 31 December 2022