User Manual for the DRIC : Diamagnetic Resisitive Interchange Code

T. Nicolas
timothee.nicolas@gmail.com

January 29, 2016

Contents

1 Introduction

The DRIC code solves a current-free eigenvalue problem for the ideal or resistive interchange mode in cylindrical geometry. In addition to resistivity, the code contains ion and electron diamagnetic effects (with possibly separate temperatures), viscosity,

perpendicular and parallel heat transport. The equations solved by the code are as follows. For more details and explanations, see Ref [1] (purely resistive equations) and Ref [2].

$$(\omega - \omega_e^\star)\psi + k_\parallel \phi + \frac{1}{1+\tau} d_i k_\parallel \frac{\beta}{2\epsilon^2} p - i\eta \nabla_\perp^2 \psi = 0 \tag{1}$$

$$(\omega - \omega_i^\star)\nabla_\perp^2 \phi + k_\parallel \nabla_\perp^2 \psi + \Omega' \frac{\beta}{2\epsilon^2} \frac{m}{r} p - i\nu \nabla_\perp^4 \phi = 0 \tag{2}$$

$$\omega p - \frac{m}{r} p'_{eq} \phi - i\chi_\perp \nabla_\perp^2 p = 0. \tag{3}$$

The code is written in python and intended to provide a very simple user friendly interface to perform parameter studies of the eigenvalues and eigenfunctions for the above three-field model. The parameters of the code are summed up in table 1

Parameter	Name in the code	Signification
β	`beta`	Central value of β
η	`eta`	Resistivity
ν	`nu`	Viscosity
$\chi_\perp$	`chiperp`	Perpendicular heat conductivity
$\chi_\parallel$	`chipar`	Parallel heat conductivity
d_i	`di`	Ion skin depth
$\tau = T_i/T_e$	`tau`	Ratio beween ion and electron temperatures
m	`m`	Poloidal mode number
n	`n`	Toroidal mode number
N	`N`	Number of radial points

Table 1: Parameters in the code

2 Dependencies

The program uses `matplotlib`, `numpy`, `scipy` and `sympy`. Make sure you have installed these dependencies.

3 Usage

3.1 Importing the program

First, import the program. A convenient way to work is the following:

- If it does not already exist, create a `Python` directory in your Home directory.
- Put an *empty* file in this directory, called `__init__.py`.
- Put this file in every subdirectory you create. This will guide `iPython`.
- For instance, if you create a subdirectory called `PROGRAMS`, which will contain different programs, and put the file `DRIC.py` inside, you will also have to copy the `__init__.py` file in this subdirectory.

- Then launch `iPython`. You should now be able to import the program with the following command

```
In [1]: from PROGRAMS import DRIC
```

3.2 Initialization

Now you can create an instance of the class `ResistiveInterchange` with the following command (assuming β=0.02, $\eta = 10^{-4}$, $\nu = 0$, $\chi_\perp = 0$, $\chi_\parallel = 0$, $d_i = 0$, $\tau = 1$, $m = 1$, $n = 1$ and $N = 999$.

```
In [2]: ric = DRIC.ResistiveInterchange(beta=0.02,
                                        eta=1e-4,
                                        nu=0.,
                                        chiperp=0.,
                                        chipar=0.,
                                        di=0.,
                                        tau=0.,
                                        m=1.,
                                        n=1.,
                                        N=999,
                                        option=1)
```

If you don't give any argument to the function, the code will ask you all the physical parameters, such as β, η, etc, as summed up in table 1. In addition, the argument `option`, which can take the values 1, 2 or 3, determines how you intend to define the pressure and iota profiles. In this case as well, if you don't provide it, the program will ask you to define it at initialization. The meaning of option is the following:

- If `option=1`, the pressure and iota profiles are defined by data points.
- If `option=2`, the pressure and iota profiles are defined by polynomial coefficients.
- If `option=3`, the pressure and iota profiles are defined by symbolic functions.

In the above command for the definition of the parameter, note that we use `m=1.`, that is, as a float and not an integer. The reason is that matematically there is no problem in regarding `m` and `n` as integers, and it makes it easier for the code to study the dependence in the mode numbers to treat them as floats.

3.3 Defining profiles

Once this step is done, you have to define the pressure and rotational transform (iota) profiles. This is done with

```
In [3]: ric.DefinePressureIotaProfiles(A_pressure,
                                       A_iota,
                                       rad_pressure=r1,
                                       rad_iota=r2)
```

The third and fourth arguments will be used only if the profiles are defined by arrays of data points (`option=1`). In this case, they define the minor radius corresponding to the data `A_pressure` and `A_iota`. If they are not provided, then it is assumed that `A_pressure` and `A_iota` are sampled simply uniformly on $[0, 1]$.

If `option=2`, the profiles are defined using polynomials. In this case the pressure and rotational transform have the following definitions:

$$p(r) = A_p[0]r^{k-1} + A_p[1]r^{k-2} + \ldots + A_p[k-1]$$
$$\iota(r) = A_\iota[0]r^{k-1} + A_\iota[1]r^{k-2} + \ldots + A_\iota[k-1],$$

where k is the length of the `A_pressure` and `A_iota`.

If `option=3`, the profiles are defined using symbolic functions. In this case after importing `sympy`, you should define a symbolic radius `r`:

```
In [4]: import sympy
```

```
In [5]: r = Symbol('r')
```

Now if you want to define the profiles of Ref. [1] for example, you can do it straightforwardly by the following command:

```
In [6]: ric.DefinePressureIotaProfiles((1-1e-3)*(1-r**2)**2+1e-3,
                                       0.461 + (1.561-0.461)*r**2)
```

3.4 Getting a purely growing seed mode

Once the initial parameters and the profiles are defined, we are good to go. Before carrying out a parameter study, it is necessary to find a first unstable mode, which will be a kind of seed for the parameter studies. This should be a purely growing mode, so it is important to use `di=0` (no diamagnetic effects) at first! You can obtain a first growth rate by the command

```
In [7]: ric.GrowthRateSearch(gamma_max,gamma_min)
```

`gamma_max` and `gamma_min` are the maximum and minimum values of the growth rate used in the search. For usual values of β, η and typical profiles, the growth rate varies typically between 10^{-1} and 10^{-3}, so it is usually good enough to use such values. If you don't provide `gamma_max` and `gamma_min`, the program will ask you to input them, with default values 10^{-1} and 10^{-2}, respectively. The program will now compute the value of ψ (or ϕ depending on the choice of parameters) in $r = 1$ for different values of `gamma`. Only when `gamma` is equal to the growth rate will this value in $r = 1$ vanish. Thus, the problem amounts to finding the zero of a function. The program computes and plots the value of the field in $r = 1$ for 100 logarithmically spaced values[1] of `gamma`, see Fig. 1.

You can see that the curve crosses the horizontal axis twice, so there are two solutions in this interval. The solutions are always located close to discontinuities of the value of the field in $r = 1$. Therefore, even if looking at Fig. 1 we are tempted to see 4 solutions, there are actually only 2, which are circled approximately on

[1] You can tell the program to use more points, for instance 500, by adding the optional argument `ntestpoints=500`, after `gamma_min`.

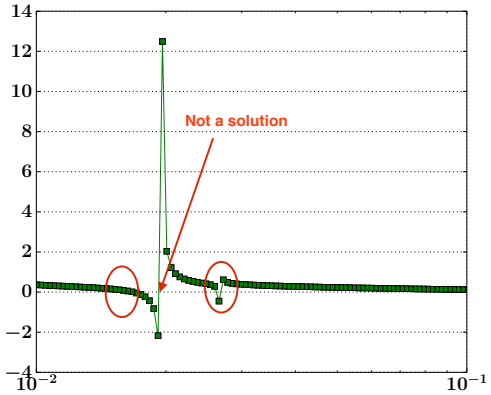

Figure 1: Plot of 100 values of the field in $r = 1$ when `gamma` varies between 10^{-1} and 10^{-2}. The approximate location of the two solutions are circled in red. An apparent crossing of the curve at a discontinuity does not correspond to a solution

the figure. One is clearly visible between 0.01 and 0.02. The second one is located beween 0.02 and 0.03. We are interested usually in the mode with the largest growth rate. Now the resolution in the scan is too coarse to be able to efficiently track down the solution, so we have to zoom. The code provides a convenient interface to do so interactively. After plotting the figure, it prints

In [8]:
```
What would you like to do ? (z,q,s)(default z)

 z = zoom on the figure and perform a new search (default)
 q = quit
 s = solve with the current provided interval (make sure it
    is appropriate)

 Your choice:
```

If you press `q` and then return, nothing will happen and you will have to start over with the search. In general, you will first have to zoom around what you assume should be close to a solution. In order to do so, you can enter `z` or simply press return since it is the default. After doing so, go to the figure and click on the figure twice, once at the left of where you think the solution is, and once at the right. Immediately, the program will recompute 100 values between these two points, and plot a new figure, as in Fig. 2. The program, again, asks what we want to do. In this case, without pressing any key, we just need to zoom on the solution by using the buttons of the python figure. We obtain Fig. 3.

Now we see that we have a figure with at the left a positive value, at the right a negative value, and in between a continuous function. That is all python needs to perform a Newton algorithm (in this particular case a variation on the same principle called brentq [3]). So now you just need to press `s` for "solve" and then return, and the solution will be obtained. The total output looks like this:

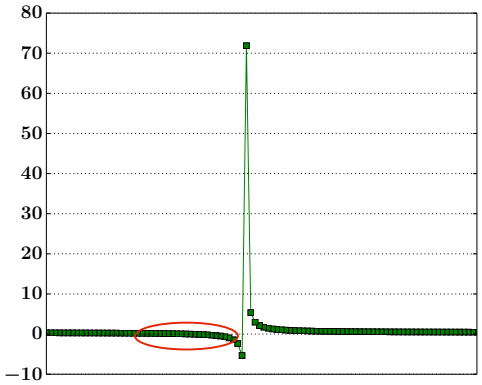

Figure 2: New computation, closer to the solution. Now we can see where the solution seems to be.

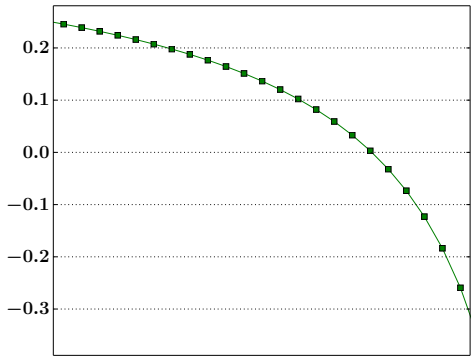

Figure 3: zoom of the previous figure

In [9]:

```
What would you like to do ? (z,q,s)(default z)

 z = zoom on the figure and perform a new search (default)
 q = quit
 s = solve with the current provided interval (make sure it
    is appropriate)

 Your choice: z
 What would you like to do ? (z,q,s)(default z)

 z = zoom on the figure and perform a new search (default)
 q = quit
 s = solve with the current provided interval (make sure it
    is appropriate)

 Your choice: s

 The solution converged
 Solution is gamma = 2.64285e-02
 Number of iterations 11
```

Important remark 1: *Notice that there is no way to be absolutely sure that there are no solution with larger growth rate in the unexplored region (in this case, for* `gamma`*>0.1). The only way to ensure it is to examine the eigenmode structure (see below how to plot the solutions). If the eigenfunction for ϕ is even around the resonant surface and does not change sign (or marginally changes sign) from $r = 0$ to $r = 1$, it is usually the mode with largest growth rate. You can also explore regions of higher* `gamma` *to make sure there is nothing there. With a combination of checks and of experience of your problem, this is usually not really an issue.*

Important remark 2: *When you use very small values of the resistivity on resistive modes (not a problem on ideally unstable modes with $\eta = 0$), the discontinuity around the solution may be so sharp that you see neither the discontinuity nor the solution on the first scan, and therefore, you may not be able to detect it. Since you may not even notice the problem, and only wonder why your eigenfunction has a weird, unusual shape, it is safer to use at first large values of the resistivity (typically, take $\eta > 10^{-6}$), and then lower η with a parameter study (see below).*

3.5 Obtaining information on the solution; Plotting it

Once you have obtained the first solution above, the most difficult part is over. All the solutions associated with your object `ric` are stored in a python list called `solutions`. At this point, it contains only one element.

In [10]:
```
ric.solutions
```

Out [10]:
```
[{'N': 999,
  'P': array([  0.00000000e+00+0.j,  -1.36518644e-02-0.j,
   -2.73037877e-02-0.j,
         ...,  -5.84469921e-05-0.j,  -1.45944953e-05-0.j,
          0.00000000e+00+0.j]),
  'Phi': array([  0.00000000e+00+0.j,   9.02899404e-05+0.j,
    1.80580812e-04+0.j,
         ...,   9.67350039e-05-0.j,   4.82862252e-05-0.j,
          0.00000000e+00+0.j]),
  'Psi': array([  0.00000000e+00+0.j,   1.83233281e-03+0.j,
    3.66466402e-03+0.j,
         ...,  -2.00199760e-03+0.j,  -1.00050000e-03+0.j,
          -1.52713249e-12+0.j]),
  'V': array([ 0.00000000+0.j,  0.00139738-0.j,
   0.00278243-0.j, ...,
         0.19563852-0.j,  0.11408916+0.j,  0.00000000+0.j]),
  'beta': 0.02,
  'chipar': 0.0,
  'chiperp': 0.0,
  'compr': 0.0,
  'di': 0.0,
  'epsilon': 0.1666,
  'eta': 0.0001,
  'm': 1.0,
  'n': 1.0,
  'nu': 0.0,
  'omega': 0.026428495880744633j,
  'sol': <scipy.optimize.zeros.RootResults at 0x107f7d6d0>,
  'tau': 1.0}]
```

If you really need to, you can explore the contents of the solution, for example you can obtain the pressure eigenfunction by the following command:

In [11]:
```
P_eigenfunction = ric.solutions[0]['P']
```

If you want to recall the value of the growth rate, you can do:

In [12]:
```
ric.solutions[0]['omega']
```

Out [12]:
```
0.026428495880744633j
```

As you can see, there is a `j` letter at the end, which means it is the imaginary part of the complex frequency. Indeed, the convention in the code is that purely growing mode is purely imaginary postive. Purely rotating in the ion (resp. electron) direction is pure real positive (resp. negative).

However, you should not need to access the data in this way in principle. Instead, in the case where you would like to work directly with the arrays of a given solution, you can get all the solution arrays for the potential ϕ, the magnetic flux ψ, the displacement ξ, the current $\boldsymbol{J}$, the pressure P, the vorticity $\nabla_\perp^2\phi$, as well as the growth rate and the radial coordinate array X. This is done with the command

In [13]:
```
X,Psi,J,Phi,Xi,P,V,omega =
    ric.GrowthRateSearch(solution=ric.solutions[0])
```

You can also directly plot the solution inside figure 1 with

```
In [14]: ric.PlotSolution(fignumber=1,solution=ric.solutions[0])
```

The result can be seen in Fig. 4. If you don't provide a `fignumber`, the result will be plotted in the figure number 1. However, it is better to open a new figure, otherwise python may not adjust the size of the figure. The optional argument `solution` specifies which solution you want to use. If you don't specify it, the last computed solution will be plotted, but it is safest to specify what solution you want to plot, after you have carried out parameter scans and you have a lot of items in the `ric.solutions` list.

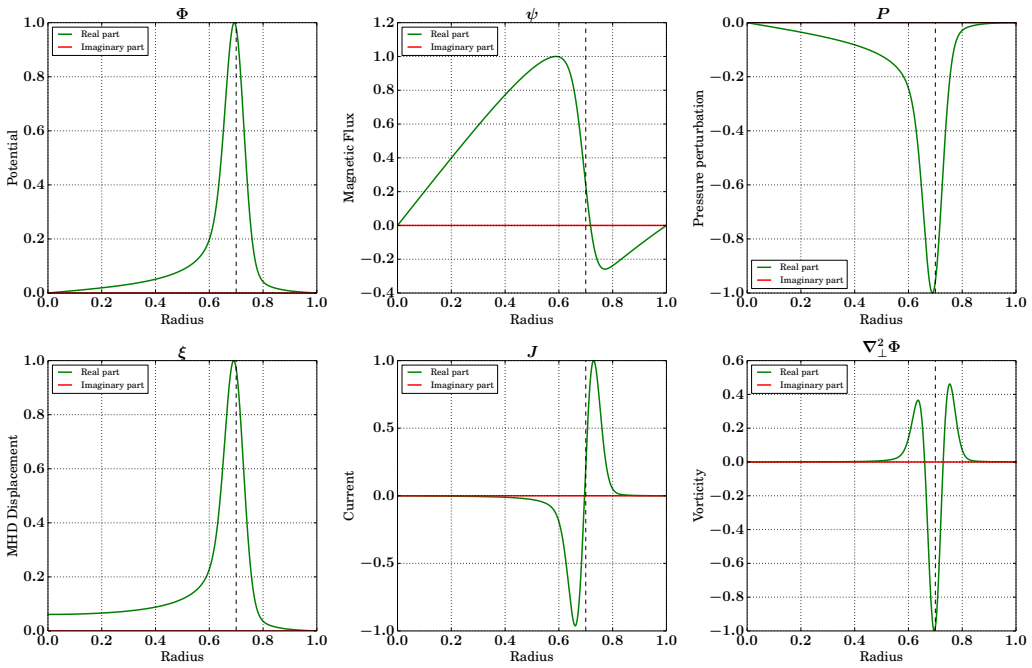

Figure 4: ϕ, ψ, ξ, $\boldsymbol{J}$, P and $V = \nabla_\perp^2 \phi$ eigenfunctions.

If you want, you can of course easily save this figure simply as a PDF (or other format allowed by python) with:

```
In [15]: matplotlib.pyplot.figure(1)
```

```
In [16]: matplotlib.pyplot.savefig('figurename.pdf')
```

3.6 Parameter scans

Now, the big advantage of this code is that it provides a very convenient and automatic way to carry out parameter scan. You can vary any parameter you like, including the mode numbers. In the initial phase above, we have obtained a first growth rate with $\eta = 10^{-4}$. Now suppose you want to know what happens for $\eta = 8 \times 10^{-5}$. You can simply launch the command:

```
In [17]: ric.ParameterScan('eta',8e-5)
```

The output will look like

```
Out [17]:
 ----------------------------------
  Trying with eta = 9.90000e-05 :
  omega = (3.7505e-23,2.6344e-02))
  ----------------------------------
  Trying with eta = 9.80000e-05 :
  omega = (-2.0630e-28,2.6259e-02)
  Low number of iterations, I increase the eta step
  Changed eta step to deta = -1.33e-06 :
  ----------------------------------
  Trying with eta = 9.66667e-05 :
  omega = (-1.4271e-28,2.6144e-02)
  Low number of iterations, I increase the eta step
  Changed eta step to deta = -1.78e-06 :
  ----------------------------------
  Trying with eta = 9.48889e-05 :
  omega = (4.4915e-28,2.5990e-02)
  Low number of iterations, I increase the eta step
  Changed eta step to deta = -2.37e-06 :
  ----------------------------------
  Trying with eta = 9.25185e-05 :
  omega = (7.6586e-28,2.5781e-02)
  Low number of iterations, I increase the eta step
  Changed eta step to deta = -3.16e-06 :
  ----------------------------------
  Trying with eta = 8.93580e-05 :
  omega = (1.3228e-29,2.5496e-02))
  Low number of iterations, I increase the eta step
  Changed eta step to deta = -4.21e-06 :
  ----------------------------------
  Trying with eta = 8.51440e-05 :
  omega = (3.9559e-28,2.5106e-02)
  Low number of iterations, I increase the eta step
  Changed eta step to deta = -5.62e-06 :
  ----------------------------------
  Trying with eta = 8.00000e-05 :
  omega = (-3.0634e-27,2.4611e-02)
  Low number of iterations, I increase the eta step
  Changed eta step to deta = -7.49e-06 :
```

You can see in real time the evolution of the growth rate when the parameter is varied. Now the complex frequency can be found easily because we have found a "seed" in the initial phase. The program simply performs a 2D solve with the `root` function [4], and adjusts automatically the step to be as efficient as possible. Sometimes, it may take larger steps than you would like (if for instance you would like to plot very smooth curves for your parameter scans). If this is the case, you can use the optional argument `maxcoeff` (default 0.1). The smaller `maxcoeff`, the smaller the relative maximum parameter step. There are other optional arguments, but they are not worth discussing here, send me an e-mail if you have some questions.

After you have completed the parameter scan, `ric.solutions` now contains many solutions:

```
In [18]: len(ric.solutions)
```

```
Out [18]: 9
```

For whatever reason, you may not be satisfied with this parameter scan, and want to reset the parameters and the state of `ric.solution` to what it was before the parameter scan, and start a new one. In that case, you have to "rewind":

```
In [19]: ric.Rewind()
```

Now you again have only one solution in `ric.solutions`. Now you may find that you want to change two parameters. For instance, you want to go toward ideally unstable modes by increasing β, and then decrease η to go toward almost ideal modes. In this case, you can perform succesively two parameter scans, like this:

```
In [20]: ric.ParameterScan('beta',0.05)
```

```
In [21]: ric.ParameterScan('eta',1e-7)
```

After you have done so, there are many solutions in `ric.solutions`:

```
In [22]: len(ric.solutions)
```

```
Out [22]: 89
```

You may now want to plot the result of the parameter scans. But you cannot plot the β and η scans on the same graph! So you have to choose which one to plot, by providing the number of the scan you are interested in. The β scan was done first, so it has the number 1. You can plot it, for instance in figure 2, by the following command:

```
In [23]: ric.PlotGrowthRatesAndFrequencies(1,fignumber=2)
```

which gives Fig. 5. There is no frequency yet, so the second part of the figure is not relevant.

If you want to plot the scan for η varying from 10^{-4} to 10^{-7} (and $\beta = 0.05$!), now this is the second parameter scan, so you can plot it by setting the first argument to 2. But you may be interested in seeing a loglog plot, since the theoretical variation of the growth rate with the resistivity is known for resistive mode. This can be done with the optional argument `log` set to `'loglog'` (you can also set it to `'semilogx'` or `'semilogy'`):

```
In [24]: ric.PlotGrowthRatesAndFrequencies(2,fignumber=2,log='loglog')
```

We obtain Fig. 6. As you can see, the growth rate curve is not a straight line, which means we have already departed from resistive mode and instead are already in the ideally unstable region.

But maybe you regret your η scan, and realize you would rather have done a parameter scan in the ion skin depth, to analyze diamagnetic effects, starting from the increased β. In this case, as in the previous case, you can also first rewind, and then perform your parameter scan in ion skin depth, for instance from 0 to 0.2:

```
In [25]: ric.Rewind()
```

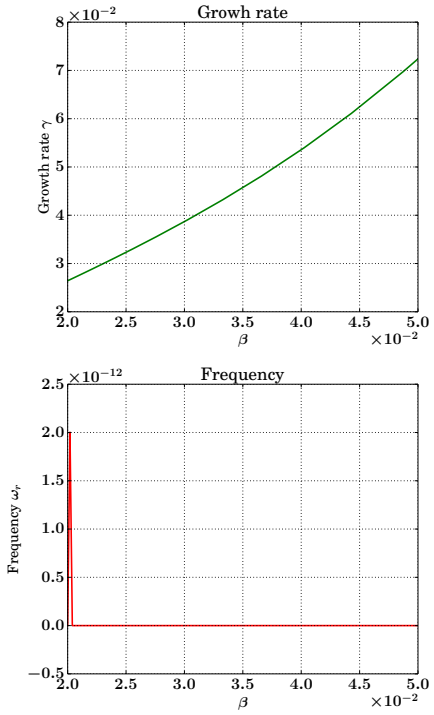

Figure 5: Scan in β, from 0.02 to 0.05

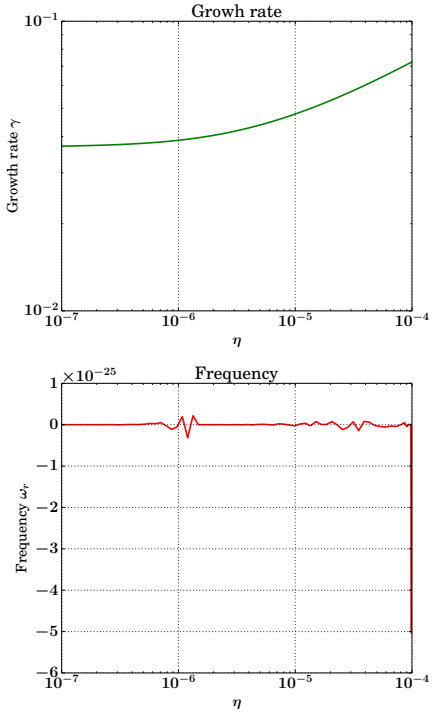

Figure 6: Scan in η, from 10^{-4} to 10^{-7}, and $\beta = 0.05$

```
In [26]: ric.ParameterScan('di',0.2)
```

If you plot the result, you now have a nice stabilization curve, as shown on Fig. 7. Notice that despite being in the ideally unstable domain, because of the significant value of the resistivity, the curve is significantly different from the curve obtained for ideal rates.

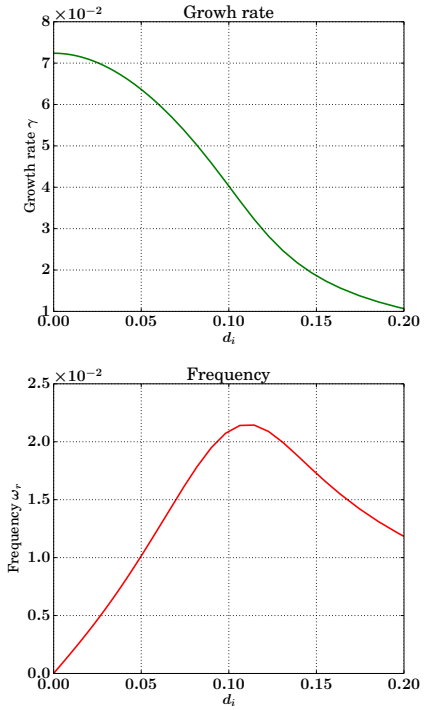

Figure 7: Scan in d_i, from 0 to 0.2, and $\beta = 0.05$, $\eta = 10^{-4}$

In many cases of interest, the resolution $N = 999$ is sufficient to reach convergence of the eigenvalues and eigenfunctions, and to resolve the sharp mode structure at the resonant surface. However, for resistive modes when the resistivity becomes very small, the mode structure becomes extremely sharp at the resonant surface, and the results can become false if the resolution is not increased. Therefore, the program automatically detects the width of the peak at the resonance, and multiplies the resolution by 2. After a few multiplication by 2, the code slows down significantly[2]. On some occasions, if the mode structure is too different from a mere peak, the program will fail computing the peak width, and the resolution will not be multiplied by 2. The program will merely say `'Failed to measure the Sheet Width'`, so you have to be careful about this kind of message. You may notice that anyway, for resistive modes when $\eta \to 0$, the law $\gamma \propto \eta^{1/3}$ works very well so it may not be so interesting to carry out parameter scans in this region.

We need to say a few words about parameter scans in the parameters m and n. As mentioned above, these parameters are treated almost exactly like the others, in the sens that they are float, and not integers in the code. The reason is that if they

[2] A way around would be to have a more flexible mesh, but this is more difficult to code.

were integers, the code would have to make big jumps from one value to an other, and would not converge. By allowing to go through real values, the code can slowly travel from, say, $m = 1$ to $m = 2$, without any problem. Thus, the only difference with the other parameters is that if you change m without changing n (and vice versa), the resonant surface moves. Thus, actually, usually the string `'m_and_n'` is used to cary a parameter scan in the mode number, so that m and n are varied proportionally all the time. For instance

```
In [27]: ric.ParameterScan('m_and_n',2)
```

performs a scan fro m to $(m, n) = (2/2)$ (if previously the $(1/1)$ mode was studied). If you still want to vary only m or only n (and hence move the resonant surface), you can do it with

```
In [28]: ric.ParameterScan('m_only',2)
```

or

```
In [29]: ric.ParameterScan('n_only',2)
```

3.7 Printing the parameters

Once you have done a few parameter scans, you may be lost and have forgotten where you are. The program provides a routine to help you remember what you did. For instance if you have performed 5 scans with the commands:

```
In [30]: ric.ParameterScan('beta',0.01)
```

```
In [31]: ric.ParameterScan('eta',9e-5)
```

```
In [32]: ric.ParameterScan('di',2e-8)
```

```
In [33]: ric.ParameterScan('m_and_n',2)
```

```
In [34]: ric.ParameterScan('m_only',3)
```

then the command `ric.PrintScanHistory()` will tell you all you did and where the data is stored in `ric.solutions`:

```
In [35]: ric.PrintScanHistory()
```

Out [35]:
```
Total number of scans: 5

 Initial parameters
------------------------------------------------------------
|beta | eta |nu|chiperp|chipar|di|tau|m | n| N |  epsilon  |
------------------------------------------------------------
|2.e-2|1.e-4|0.|  0.   |   0. |0.| 1.|1.|1.|999|1.66600e-01|
------------------------------------------------------------

 Scan No 1: beta was varied from 2.000e-02 to 1.000e-02.
    Scan contained in solutions[0] to solutions[13]
 Scan No 2: eta was varied from 1.000e-04 to 9.000e-05. Scan
    contained in solutions[13] to solutions[27]
 Scan No 3: di was varied from 0.000e+00 to 2.000e-08. Scan
    contained in solutions[27] to solutions[36]
 Scan No 4: m and n were varied together from 1.000e+00 to
    2.000e+00. Scan contained in solutions[36] to
    solutions[52]
 Scan No 5: m was varied from 2.000e+00 to 3.000e+00. Scan
    contained in solutions[52] to solutions[67]
```

You can also print the parameters of any solution, for instance, the ith solution, with `ric.PrintParameters()`:

In [36]:
```
ric.PrintParameters(solution=ric.solutions[i])
```

If you don't provide any optional argument, the last used parameters will be used.

3.8 Saving Data

The routine to save the data is pretty straightforward. Simply do

In [37]:
```
ric.Save(filename='your_file_name')
```

Don't use any extension to the filename. It will automatically add the extension `.npz`. If you don't provide any filename (if you simply use `ric.Save()`), then an automatic filename will be generated based on the inital and last parameters. For instance, for the case of the 5 parameter scans above, this filename would be:

In [38]:
```
beta_from_2.0e-02_to_1.0e-02_eta_from_1.0e-04_to_9.0e-05_
nu_0.0e+00_chiperp_0.0e+00_chipar_0.0e+00_di_from_0.0e+00_
to_2.0e-08_tau_1.0e+00_m_from_1.0e+00_to_3.0e+00_
n_from_1.0e+00_to_2.0e+00_N_999_epsilon_1.7e-01.npz
```

It is very long due to the quantity of information. In spite of this, it does not contain all the information since only the initial and last parameters are indicated, not the path to go there. If there is already a file with the same name, the program will ask you if you want to replace it. Thus, you are free to use or not use your own filenames.

3.9 Loading Data

If you have saved data in the past in an `'.npz'` file, you can be tempted to simply load it, in this way:

```
In [39]:  Data = np.load('filename.npz')
```

If you do this, you will be able to access the data, and even plot it with your own routines if you figure out the structure of the data, which is not too complicated, but you will not be able to make more parameter scans, or to use any of the methods we have described above. Therefore it is not the good method to proceed. The good method is to use the same method as in the initialization, that is, `DRIC.ResistiveInterchange()`, except in this case, instead of providing no argument, or the physical parameters plus `option` as argument, you should provide the file name 'filename.npz' as argument. Therefore, you can recover all the structure of the python object by the following command:

```
In [40]:  ric = DRIC.ResistiveInterchange(filename='filename.npz')
```

Once the data is loaded in this way, you can restart working exactly like before you saved the data, perform new parameter scans, visualize the data, etc.

4 Summary of the commands

In this last section, we summarize the different commands we have reviewed. For every routine, we indicate the mandatory arguments in the parentheses, and then describe the optional arguments below.

4.1 Initialization

```
In [41]:  ric = DRIC.ResistiveInterchange()
```

Optional arguments:

- `beta` : the value of the central β
- `eta` : the value of η, the resistivity
- `nu` : the value of ν, the viscosity
- `chiperp` : the value of $\chi_\perp$, the perpendicular heat conductivity
- `chipar` : the value of $\chi_\parallel$, the parallel heat conductivity
- `di` : the value of d_i, the ion skin depth (multiplies both ion and electron diamagnetic terms)
- `tau` : the value of $\tau = T_i/T_e$
- `m` : the value of m, the poloidal mode number
- `n` : the value of n, the toroidal mode number
- `N` : the value of N, the number of points in the radial direction
- `option` : How the profiles will be defined ($1\longrightarrow$ by data points, $2\longrightarrow$ by polynomial coefficients, $3\longrightarrow$ by symbolic functions)
- `filename` : the filename to load the data from. If it is used, then all the other options are ignored.

4.2 Defining the profiles

```
In [42]: ric.DefinePressureIotaProfiles(A_pressure,A_iota)
```

Optional arguments:

- `rad_pressure` : if `option=1`, the radial positions corresponding to the array `A_pressure`. Ignored otherwise. If `option=1` and this is not provided, it is assumed to be equal to `numpy.linspace(0,1,len(A_pressure))`.
- `rad_iota` : if `option=1`, the radial positions corresponding to the array `A_iota`. Ignored otherwise. If `option=1` and this is not provided, it is assumed to be equal to `numpy.linspace(0,1,len(A_iota))`.

In cases `option=1` and `option=2`, `A_pressure` and `A_iota` can be lists or numpy arrays. See section 3.3 for the link between the polynomial coefficients `A_pressure`, `A_iota`, and the actual profiles. If `option=3`, you must first define a symbolic radius with

```
In [43]: r = Symbol('r')
```

and then define `A_pressure` and `A_iota` as symbolic functions of `r`.

4.3 Performing a growth rate search ($d_i = 0$)

```
In [44]: ric.GrowthRateSearch()
```

Optional arguments:

- `gamma_max` : Maximum growth rate for the search (default 10^{-1})
- `gamma_min` : Minimum growth rate for the search (default 10^{-1})
- `ntestpoints` : Number of test points in the search (default 100, usually sufficient and fast)

4.4 Get a particular solution

```
In [45]: X,Psi,J,Phi,Xi,P,V,omega = ric.GrowthRateSearch()
```

Optional arguments:

- `solution` : If this argument is not provided, the last computed solution is used. Otherwise,it can be, for instance, `ric.solutions[10]`, which means the 10^{th} computed solution.

4.5 Plot the solutions

```
In [46]: ric.PlotSolution()
```

Optional arguments:

- `fignumber` : Number of the figure to plot in
- `solution` : If this argument is not provided, the last computed solution is used. Otherwise,it can be, for instance, `ric.solutions[10]`, which means the 10^{th} computed solution.

4.6 Perform parameter scans

```
In [47]: ric.ParameterScan(parameter_name,parameter_goal_value)
```

- `parameter_name` is a string which can have the following values
  - `'beta'`
  - `'eta'`
  - `'nu'`
  - `'chiperp'`
  - `'chipar'`
  - `'di'`
  - `'tau'`
  - `'m_and_n'`
  - `'m_only'`
  - `'n_only'`
- `parameter_goal_value` can be set freely

Optional arguments:

- `maxcoeff` : if you make it smaller (default 0.1), then smaller steps will be taken even if the solver can converge to the gola value more easily. This will smoothen your curves if you want to have many points.

4.7 Plot the growth rate and frequency as a function of the parameter scan

```
In [48]: ric.PlotGrowthRatesAndFrequencies(scan_number)
```

`scan_number` is the number of the scan, which you would like to plot.
Optional arguments:

- `fignumber` : The number of the figure to plot in
- `log` : Determines the log type of the figure (the log type in the vertical axis is applied only to the growth rate part because the frequencies can easily be negative). The values of this field can be `'loglog'`, `'semilogy'` or `'semilogx'`.

4.8 Rewind

```
In [49]: ric.Rewind()
```

There are no arguments. This will go back in time, erase the last scan and reset the parameters to what they were at the beginning of the last scan.

4.9 Print history of the past scans

```
In [50]: ric.PrintScanHistory()
```

There are no arguments. This will print on the screen a summary of all the last scans. This is very helpful in knowing how the parameters have changed, and where the data you are interested in is stored.

4.10 Print the parameters of a given solution

```
In [51]: ric.PrintParameters()
```

Optional arguments:

- `solution` : If this argument is not provided, the last computed solution is used. Otherwise, it can be, for instance, `ric.solutions[10]`, which means the 10^{th} computed solution.

4.11 Saving the data

```
In [52]: ric.Save()
```

Optional arguments:

- `filename` : The name of the file to save in (without extension). If not provided, an automatic file will be created. You will be warned ans asked before an existing file is erased.

References

[1] R. Ueda, *et al.* Phys. Plasmas **21(5)**, 052502 (2014).

[2] T. Nicolas *et al.* Nucl. Fusion **56(2)**, 026008 (2015).

[3] scipy.optimize.brentq. http://docs.scipy.org/doc/scipy-0.13.0/reference/generated/scipy.optimize.brentq.html.

[4] scipy.optimize.root. http://docs.scipy.org/doc/scipy-0.13.0/reference/generated/scipy.optimize.root.html.