EXERCISES WITH THE STIC CODE
SOLAR SPECTROPOLARIMETRY AND DIAGNOSTIC TECHNIQUES

J. de la Cruz Rodriguez, Stockholm University, Albanova University Centre, SE-10691 Stockholm, Sweden.

Description of the STiC code (manuscript): http://dubshen.astro.su.se/~jaime/stic.pdf.
Installation instructions: http://dubshen.astro.su.se/~jaime/2018_HAO_school/stic_compile.txt
Source code: http://www.github.com/jaimedelacruz/stic/

1 Synthesis from a model atmosphere

The idea is to perform a 1 pixel synthesis using STiC. The students will get experience manipulating the input files and setting up LTE and nLTE lines which is identical to RH. Once the synthesis is done, we will use the python tools to read the output from STiC.

The key files in this part are:

- **input.cfg**: the main input/output file where file names and node placement are specified.
- **keyword.input**: the RH control file. Most of the keywords from RH work, except I/O related ones.
- **atoms.input**: active (nNLTE) atoms and passive atoms that are used to compute the background opacity.
- **kurucz.input**: here we can specify LTE lines in Kurucz format.

The first step is to create the input model atmosphere. We have included a commented python script were the steps are described. In this case we will read the FALC model from a formatted text file and we will store it in a STiC model container. The python tools that we have included have a *model* object class that can be used to read an existing model from HD or to create an empty container.

The *prepare_input_model.py* script will fill in the relevant variables of the STiC:

```
import sparsetools as sp
import numpy as np

# Load FALC from a text file into a variable
fal = np.loadtxt('falc_cgs.txt')
ndep, nvar = fal.shape

# Create container with 2x2 pixels
m = sp.model(nx=2,ny=2,ndep=ndep,nt=1)


# Fill in variables
for zz in range(ndep):
    m.z[0,:,:,zz]    = fal[zz,0]
    m.temp[0,:,:,zz] = fal[zz,1]
    m.vlos[0,:,:,zz] = fal[zz,2]
    m.vturb[0,:,:,zz]= fal[zz,3]
    m.pgas[0,:,:,zz] = fal[zz,4]
    m.nne[0,:,:,zz]  = fal[zz,5]

# Let's also assume that we have a
# constant magnetic field (with depth)
m.Bln[:] = 400.
m.Bho[:] = 650.
m.azi[:] = 19.0*3.1415926/180

# Save model
m.write('FALC.nc')
```

Now we can execute the python script to create actual model and then we can modify our files as we wish.

1.1 input.cfg

Let's consider a case where we want to synthesize spectral lines that can be observed with NASA's IRIS satellite, and with the Swedish 1-m Solar Telescope. That case could include the Mg II h&k, Ca II H&K lines, Ca II $\lambda 8542$ and Fe I $\lambda 6301$ & $\lambda 6302$. The assumed case contains all cases that we can encounter with STiC: non-LTE/PRD lines, non-LTE/CRD lines, LTE lines.

By default, RH will define a wavelength grid that is specified in the atom files in order to solve the non-LTE problem. But we might want our output in a different wavelength grid than the one set by RH. Therefore, the **input.cfg** is used to define regions where the emerging spectra will be computed and saved to the output file. The standard way of defining a region is as follows:

```
region = w0, dw, nw, scl, instrument_type, \
   instrument_calibration_file
```

where:

- $w0$ is the first wavlength of the region [Å].
- dw is the wavlength step [Å].
- nw is the number of wavelength points
- scl is a scaling factor. The synthetic spectra are computed in CGS units. But if you have normalized your observations by a factor (e.g., the quiet-Sun average continuum intensity), you can apply the same correction to the synthetic spectra on the fly. This parameter is mostly used with inversions. To work in CGS units just set it to 1.
- *instrument_type* is the spectral degradation that we want to apply to our region. It takes values: $none$, $spectral$, fpi (not supported).
- *instrument_calibration_file* is the name of a file with the instrumental psf and other parameters that could be needed by the code. In the case of spectral, it should contain a PSF with odd number of elements and not more elements than nw.

And example of a few regions could look like this:

```
region = 6301.1, 0.02, 41, 1, none, none
region = 6302.1, 0.02, 41, 1, spectral, psf.nc
```

If more than one region is specified in the input file, the code will concatenate them in the same order as they appear in the input file.

Other important keywords that can influence the synthesis are:

```
input_model =  modelin.nc
output_profiles = synthetic.nc
keep_nne = {0|1}
recompute_hydro = {0|1}
```

The *keep_nne* keyword will preserve the input electron density, otherwise the electron density will be recomputed assuming LTE. You can set the *recompute_hydro* keyword if you want to set the atmosphere to hydrostatic equilibrium. In the latter case, the integration will be started using the provided value of P_{gas} at the top of the atmosphere.

1.2 atoms.input

This file usually looks like this:

```
 # Nmetal
   12

# Metals
  Atoms/H_6.atom    PASSIVE LTE_POPULATIONS
  Atoms/C.atom      PASSIVE LTE_POPULATIONS
  Atoms/O.atom      PASSIVE LTE_POPULATIONS
  Atoms/SiIV.atom   PASSIVE LTE_POPULATIONS
  Atoms/Al.atom     PASSIVE LTE_POPULATIONS
  Atoms/CaII.atom   ACTIVE  ZERO_RADIATION
  Atoms/FeI.atom    PASSIVE LTE_POPULATIONS
  Atoms/He.atom     PASSIVE LTE_POPULATIONS
  Atoms/MgII.atom   ACTIVE  ZERO_RADIATION
  Atoms/N.atom      PASSIVE LTE_POPULATIONS
  Atoms/Na.atom     PASSIVE LTE_POPULATIONS
  Atoms/S.atom      PASSIVE LTE_POPULATIONS
```

The atoms that are marked as active will be treathed in nLTE, whereas the passive atoms will be used in the computation of background opacity. If we remove all those passive atoms, you might find that you are lacking opacity in the continuum, especially in the UV. For active atoms, `ZERO_RADIATION` is usually a better initial solution than LTE.

1.3 keyword.input

We have provided a default file that is well commented. We will not go through all the keywords here but we will mention a few that could affect your inversions. The most important ones are those that control the Jacobi iteration scheme:

```
N_MAX_ITER = 100
ITER_LIMIT = 6.0E-5
NG_DELAY   = 15
NG_ORDER   = 2
NG_PERIOD  = 7
PRD_N_MAX_ITER = 2
PRD_ITER_LIMIT = 1.0E-4
```

The listed values are taken from a config file that is normally used for inversions. If we are operating in inversion mode, we will be using finite-difference based derivatives of the intensity, and therefore we need to converge the atom population densities more than we would normally do for a regular forward calculation. But for synthesis mode we can be less restrictive with the convergence ratio (e.g., $\mathrm{ITER\_LIMIT} = 1.0 \times 10^{-3}$).

NG acceleration will speed up our convergence significantly. But it cannot be applied too early. Otherwise the problem becomes numerically unstable and the populations will not converge. Also, if the period is too short it will not help either. The optimal set of values depends on the intrinsic properties of the atmosphere. We found that with typical setups, these values work, but if you see many errors related to singular matrix, it is better to increase the NG_DELAY and NG_PERIOD keywords.

1.4 Exercise instructions

The student is expected to perform the following tasks:

1. Create the input model atmosphere using the provided prepare_input_model.py. Make sure you understand what the script is doing. Try to plot some of the variables from the model using matplotlib!

2. Edit input.cfg and add the regions that you want to compute the central wavelength of the aforementioned lines are provided in Table 1.4. Make sure that mode = 2 (synthesis).

3. Select your active atoms in atoms.input

4. Open the kurucz.input file and make it point to Atoms/kurucz_6301_6302.input so we can include the LTE lines that we discussed above.

5. Execute the code with at least 2 cores and wait until it finishes.

6. plot the result with plot.py.

line	λ_0	Instrument
Mg II k	2795.528	IRIS
Mg II h	2802.705	IRIS
Ca II K	3933.664	CHROMIS
Ca II H	3968.469	CHROMIS
Ca II $\lambda 8542$	8542.091	CRISP/IBIS/gFPI
Fe I $\lambda 6301$	6301.501	CRISP/IBIS/gFPI
Fe I $\lambda 6302$	6302.493	CRISP/IBIS/gFPI

If everything went fine, you should get something like this:

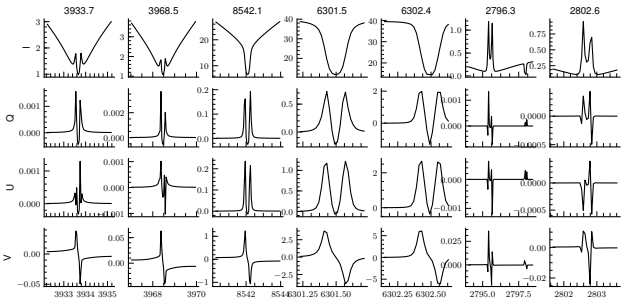

Figure 1: *Example of a synthesis result. The intensity units are [nW m^{-2} Hz^{-1} sr^{-1}].*

2 Inversion of a synthetic profile

In exercise 1 we synthesized spectra in several lines, using the FALC atmosphere. We can try to perform an inversion on those profiles, after adding some noise to our synthetic observation. This exercise will show that even when all our assumptions in the radiative transfer and physics are correct, there is only so much information we can extract from our observations.

How is the agreement between the FALC model and the output from the inversion? What could not be recovered? Can you guess why?

2.1 Exercise instructions

1. First execute the python script that will read the output profiles from exercise 1 and it will add noise to those profiles, creating a synthetic observation with noise.
2. Run STiC and let the inversion finish.
3. Plot the result with plot.py.

In my case, the result looks like Fig. 2.

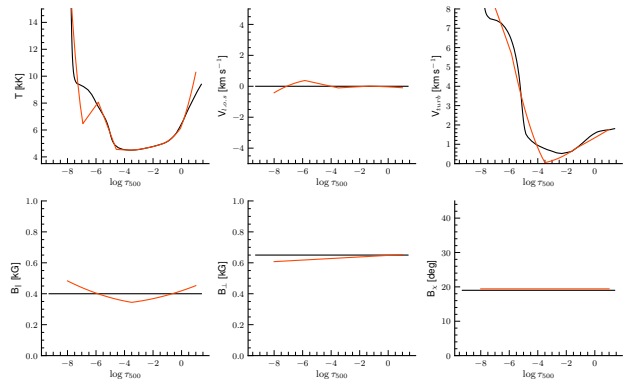

Figure 2: *Results from the inversion of our synthetic spectra from the FALC model.*

3 Inversion of an observation in CRD

Now that we have some control of the input files, we will try to invert an example observation in the Ca II $\lambda 8542$ line. This example was acquired with the CRISP imaging spectropolarimeter at the Swedish 1-m Solar Telescope. The subfield of 2×2 pixels was extracted from the penumbra of a sunspot.

The wavelength sampling of this observation is not regular, but all the line positions were chosen to be multiples of 35 mÅ. This is convenient because during the inversion we will need to convolve all synthetic spectra with the CRISP spectral PSF. This convolution is done with FFT's, and for that we need to have regularly sampled data. The trick is to generate an input data file that includes extra wavelength points, even if we have not observed them (see Fig. 3). STiC (RH) will compute the spectra in those extra points and they will be used by the convolution routine, but they will not be accounted in the inversion.

The fine grid should (at least) be as small as half the spectral resolution of the instrument (remember the Nyquist sampling theorem). The instrumental profile of CRISP has a FWHM of approximately 105 mÅ, so 35 mÅ clearly satisfies that threshold. Fig. 4 shows the CRISP transmission profile at 854 nm. If the denser grid is chosen too coarse, the first points right besides the central point would have an value below 0.5 and the profile would start looking like a delta function.

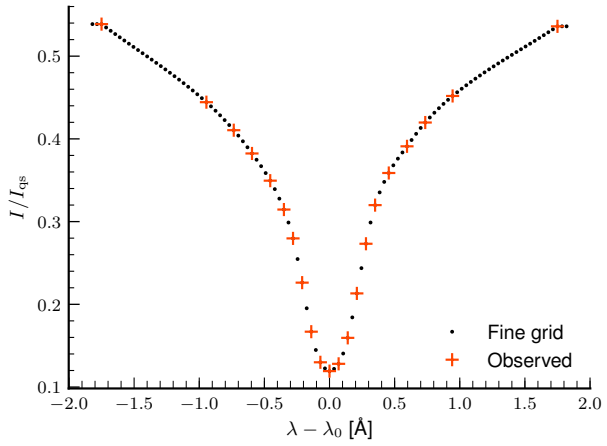

Figure 3: *Observed profile with irregular sampling (red). The black dots mark a finer and regular grid of wavelength points that contains all observed points. The synthetic spectrum will be computed in the fine grid, but only the observed points will contribute to χ^2.*

In this exercise we have prepared a script that will format the data. It will compute the fine grid of points, and copy the observed data to the corresponding line positions in the data. The array where we tell the code the noise of each data point is adjusted to that the extra points included in the data have effectively a zero weight. The latter is done by increasing the noise to a very large value. Additionally we will artificially decrease the noise estimate of Stokes Q, U and V in order to make them have a significant imprint in the penalty function.

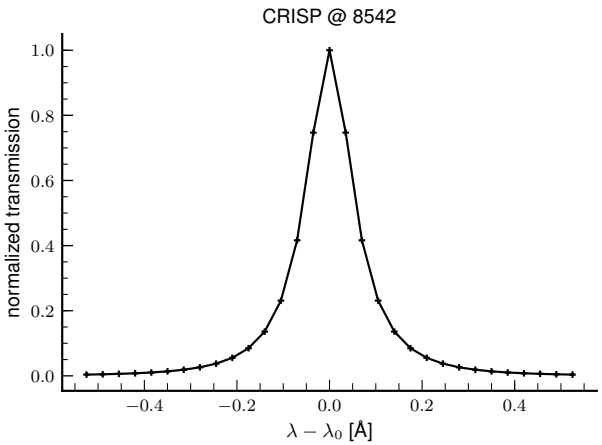

Figure 4: *Transmission profile of CRISP at 854.2 nm.*

3.1 Exercise instructions

1. Open the python script that prepares the data (*prepare_data.py*). Can you understand the part where the finer grid is created?
2. When you run the script, it will print a line with the *region* information for *input.cfg*. You should copy that line in your input file.
3. If you look into *Atoms/CaII_bklm.atom* you will see that some of the transitions (the K&H lines) are computed in PRD by default. But we are only interested in the $\lambda 8542$ line, which can be computed in CRD. We can disable PRD calculations in *keyword.input*, just make sure that the keyword `PRD_N_MAX_ITER = 0`. Make sure PRD is disabled in this case, otherwise the inversion will be significantly slower.
4. The observation has 2×2 pixels. You can test running the MPI version of the code by executing it with more than one slave processor, e.g.: mpiexec -n 4 STiC.x
5. Once the inversion is finished, you can check the results using *plot.py* it will generate one plot with the fits and the inferred model stratification for each pixel.

Some questions that you can ask yourself when you look at the results:

- What happens when you perform the inversions with the default setup?
- Are the fits in all Stokes parameters equally good?
- What happens with the inversion results if you modify *prepare_data.py* and decrease the noise estimate in Q& U by a factor $\times 10$ and in V by a factor $\times 4$? Are the fits better?

In the latter case, with the extra weight in Stokes Q, U and V, I get the results illustrated in Fig. 5.

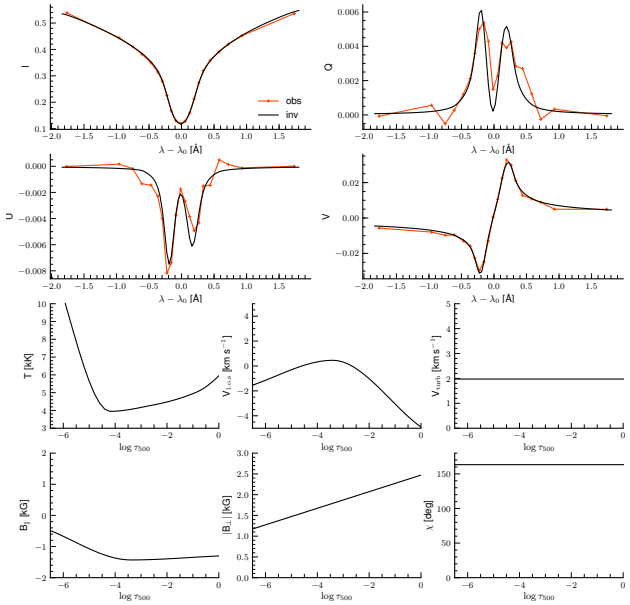

Figure 5: *Results from the inversion of the first pixels of the FOV. The fits to the observations are displayed in the upper part of the figure, whereas the stratification of the inferred model is shown in the lower half of the figure.*

4 Inversion of an observation in PRD

We will try to invert the example pixel that is included with STiC, but with a bit more insight than just running the example. This pixel is extracted from a CRISP/CHROMIS observation at the Swedish 1-m Solar Telescope. The main aim of this exercise is to show the students what happens when we invert an observation with different setups. The students will run the example with the following node configurations.

	T	$v_{l.o.s}$	v_{turb}	$B_{\|}$	$B_{\perp}$	B_{χ}	α
Set 1	7	4	4	2	2	1	0.0
Set 2	13	7	7	4	3	2	0.0

The students will have to modify the scripts to accommodate the number of nodes for each case. We have included a plot script to show the results. We should make sure that PRD iterations are activated in *keyword.input*, by setting `PRD_N_MAX_ITER = 2` (or larger).

The students should discuss the following questions:

- What happens with the fit quality when the number of nodes is increased?
- What happens with the model atmosphere when the number of nodes is increased?

4.1 Exercise instructions

1. Edit input.cfg and let STiC perform the inversion with set 1.

2. Edit again input.cfg and let STiC perform the inversion with set 2. Use a different output atmosphere/profile files.

3. Make the plots with plot.py (you will need to edit the file to read each of the results).

4. Try to answer the questions above.

Now there is a twist in this exercise. Mathematically it is possible to tell the code that we *know* that certain solutions are (probably) not physically relevant. For example, we know that our atmosphere does not behave like white noise, so there should be some form of continuity in the stratification of the different parameters. We could also argue that those wiggles that appear when we increase the number of nodes are not needed, as long as we can still fit the observations.

STiC allows including regularization terms in the Levenberg-Marquardt algorithm. To do so, we have some keywords in the input file:

```
regularize = 1.0
regularization_type = 1,1,1,1,1,1,1
regularization_weights = 1,1,1,0.1,0.1,0.1,1
```

The *regularize* keyword is a global weight that sets how much regularization should be applied in general (α in our table). When set to zero, no regularization is applied. The *regularization_type* keyword is a coma-separated chain with whe regularization type used for each parameter. Use the following numbers to penalize as: (1) Tikhonov (1^{st} derivative), (2) deviations from the mean, (3) deviations from zero, (4) second derivative, (5) applies (3)+(4). Note that (3) and (5) can only be applied to microturbulence. The order of the chain is: temperature, $v_{l.o.s}$, v_{turb}, $B_{\parallel}$, $B_{\perp}$, B_{χ}.

Similarly, we can have a relative scaling between the regularization applied to each parameter, which is done by adjusting the chain in *regularization_weights*. The order of the variables in the chain is the same as before.

We will try to re-run set 2, but this time we will add regularization. Try to make the inversion for the values $\alpha = 1.0$ and $\alpha = 10.0$. How does the result change? In the default run I get the result in Fig. 7.

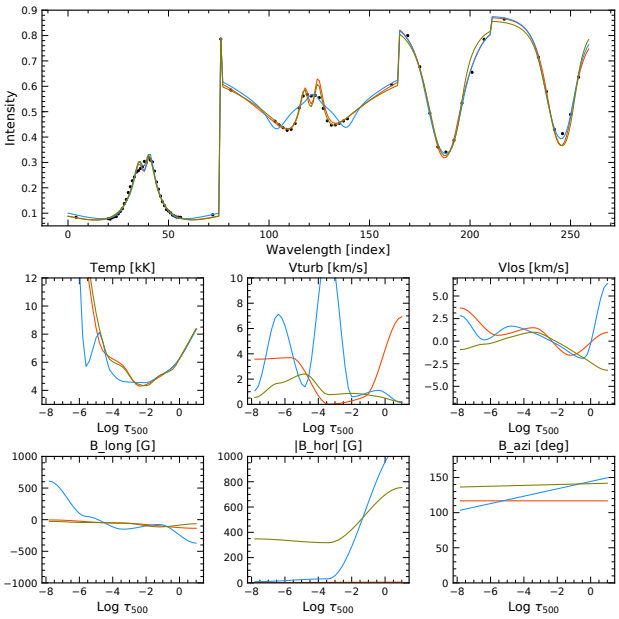

Figure 6: *Inversion results for the example pixel included with STiC. The red curve is the standard setup without regularization. The blue curve corresponds to an inversion without regularization and many more nodes. The olive curve is a fully regularized inversion with the same setup as the blue curve.*

5 Computing the response functions of your model

STiC allows computing numerically response functions of a given model. We can choose which parameters we want to include in these calculations. The main keywords that we need to change in input.cfg are:

```
get_response = 1,1,1,1,1,1
mode = 4
```

In this case, we will set to 1 or to 0 the variables for which we want to compute the response function. The order is as usual: T , $v_{l.o.s}$, v_{turb}, $B_{\parallel}$, $B_{\perp}$, B_{χ}. Please remember to specify as `input_model` the result of your inversion, otherwise the calculation will be done with the wrong model. The response functions are stored in the output *profiles* file, and the variable in the file is called *derivatives*. The latest version of sparsetools.py in the GitHub repository will read the response functions automatically if they are present in the file. The variable name in the profile container is *rf*. The latter has dimensions [nt, ny, nx, nRF, ndep, nw, nStokes].

For example, if we want to read and plot the response function for temperature, we would to something like this:

```
import sparsetools as sp
import matplotlib.pyplot as plt
import imtools as im

# Load the synthetic spectra and the RFs and
  the input model
p = sp.profile('synthetic.nc')
```

```
m = sp.model('FALC.nc')

# make the plot of the first pixel
xx = 0; yy = 0; ivar = 0
extent = (0, p.nw, m.ltau[0,yy,xx,-1], m.ltau
  [0,yy,xx,0])

plt.imshow(im.histo_opt(p.rf[0,yy,xx,ivar
  ,::-1,:,0]), cmap='gist_stern', extent=
  extent, aspect='auto')
plt.ylabel(r'log $\tau_{500}$')
plt.xlabel(r'Wavelength array index')
plt.show()
```

Since RFs measure the rate at which the intensity changes per unit of parameter, the RF's are somehow intrinsically scaled by the actual intensity of the profile. If your line is located in a spectral region where the Sun does not produce many photons, then your RF will also look weaker than that from a line in a different brighter region. In this example the Mg II lines appear much darker than all the other considered lines for this reason.

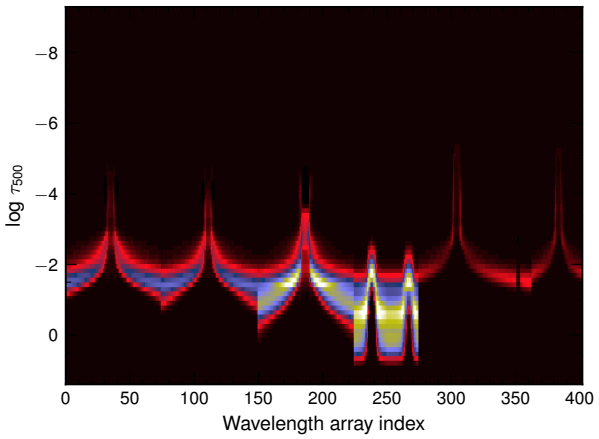

Figure 7: *Response functions to temperature as a function of wavelength. The color scale is highly non-linear, but it allows visualizing a large dynamic range like the one in this image.*

A Included python tools

A.1 The model class

The STiC python tools can be found in the distribution in stic/pythontools/py2/sparsetools.py. Let's assume that we want to create a model container with dimensions $nx = 2, ny = 3, ndep = 45, nt = 1$. The object will be created with the following numpy arrays inside: z, cmass, ltau, temp, vlos, vturb, Bln, Bho, azi. The dimensions of these variables will be $[nt, ny, nx, ndep]$, where $ndep$ is the fast axis.

```
import sparsetools as sp

# Create empty container
m = sp.model(nx=2, ny=3, ndep=45, nt=1)

# Or read an existing model from HD
m = sp.model('modelin.nc')
```

Let's assume that we have loaded data from a 3D rMHD simulation into variables named $temp$, z, v_z, P_{gas}, P_e, B_z, B_y, B_x. , which have the same dimensions as in the previous example. In order to copy these variables to our data container we can simply do:

```
import sparsetools as sp
import numpy as np

# Create empty container
m = sp.model(nx=2, ny=3, ndep=45, nt=1)

# Copy the data
m.temp[:,:,:,:] = temp
m.z[:,:,:,:]    = z
m.vlos[:,:,:,:] = vz
m.Bln[:,:,:,:]  = bz
m.Bho[:,:,:,:]  = (bx**2 + by**2)**0.5
m.azi[:,:,:,:]  = np.arctan2(by, bx)

# We need to provide at least one gas
 pressure/density variable:
m.pgas[:,:,:,:] = pgas
m.pe[:,:,:,:]   = pe

# ...or if we had density:
m.rho[:,:,:,:] = rho

# we can also set the microturbulence to 2 km
 /s (optional):
m.vturb[:,:,:,:] = 2.0e5

# Now we can write the data to HD
m.write('model_atmosphere.nc')
```

A.2 The profile class

Similarly, we also have included a *profile* class that will be used for I/O operations with spectral profiles. The profile class contains the following variables $wav[nwav]$, $dat[nt, ny, nx, nwav, nStokes]$, $weights[nwav, nStokes]$. The $weights$ variable is only used in inversion mode and in reality it represents the noise estimate of each data point. Obviously the noise can be the same for all wavelength points, but we can use this variable to mask certain wavelength regions (e.g., telluric lines) or to give extra weight to certain spectral windows. In synthesis mode these values are ignored.

We can copy our observed data to a profile container as follows. Imagine that we have loaded our observations in python with names I, Q, U, V, wav.

```
import sparsetools as sp

# Create empty container
obs=sp.profile(nwav=201,nx=2,ny=3,nt=1,ns=4)

# Fill in the data
obs.wav[:] = wav
obs.dat[0,:,:,:,0] = I
obs.dat[0,:,:,:,1] = Q
obs.dat[0,:,:,:,2] = U
obs.dat[0,:,:,:,3] = V

# And optionally the noise level
# In this case let's assume that the
# observations are normalized to the
# average quiet-Sun continuum
obs.weights[:,:] = 1.e-3

# and write the dataset to HD
obs.write('observation.nc')
```

If we have and existing dataset, and we want to extract a spectral window from it, we can use the method *extractWav*:

```
# Note that w1 is not included in the
# range but w1-1, like in numpy
obs1 = obs.extractWav(w0=0, w1 = 100)
```

We can also extract part of the field-of-view, for all wavelengths:

```
obs1 = obs.extract(x0=0, x1=1, y0=0, y1=1)
```

Or compute the mean spectrum over the FOV:

```
SPmean = obs.averageSpectrum()
```