Simulation data handling

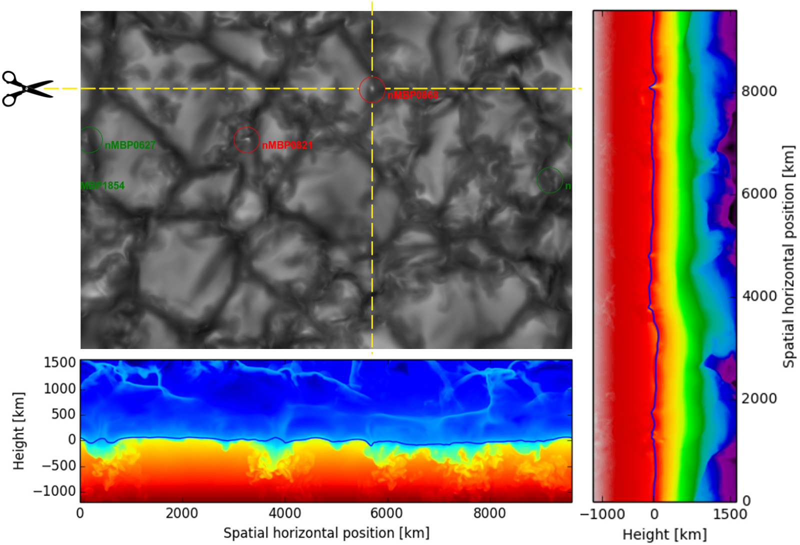
Flavio Calvo

Institute for Solar Physics, Stockholm's University, Sweden

From virtual to real observations,
SOLARNET-FoMICS school on spectropolarimetry
11 October 2019

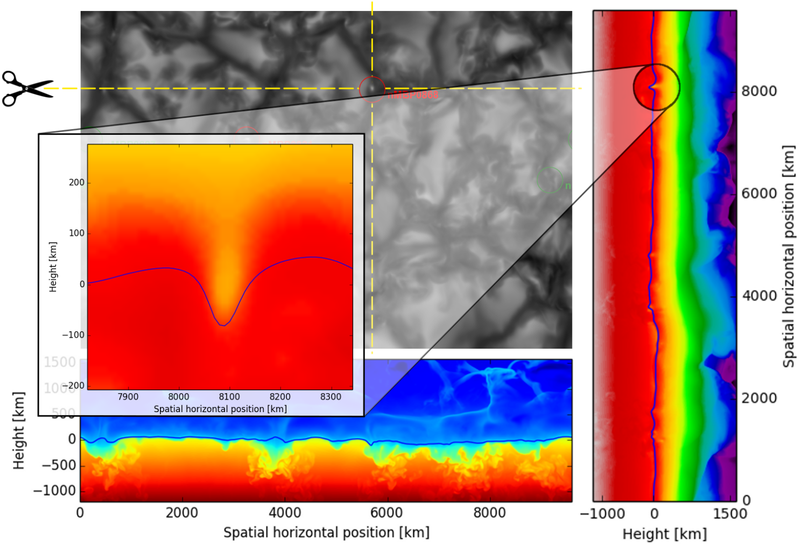
Contents

Why making simulations?



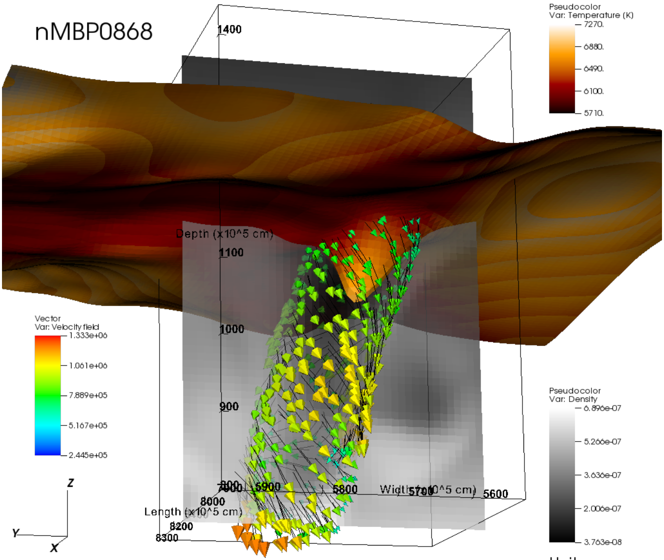
Emergent intensity (top left), temperature (bottom), log(rho) (right)

Why making simulations?



Emergent intensity (top left), temperature (bottom), log(rho) (right)

Why making simulations?



Units: cgs

Why making simulations?

We run simulations. . .

- to “see” inside of the Sun and learn from it
- to verify our knowledge of the solar atmosphere by confronting to observations
- to make predictions

However, we have some drawbacks:

- The data provided by simulations cannot be directly confronted to observations
- We are not “observing” the real Sun, but merely a *numerical construction* of it!
- Only a small parcel of the Sun can me simulated with present computers
- Box-in-a-star simulations require specific boundary conditions that poorly represent the dynamics outside of the simulation box

Ideal radiative MHD equations

$$\begin{aligned}
\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) &= 0, \\
\frac{\partial (\rho \mathbf{v})}{\partial t} + \nabla \cdot \left(\rho \mathbf{v}\mathbf{v} + \left(P + \frac{\mathbf{B} \cdot \mathbf{B}}{2} \right) \mathbf{I} - \mathbf{B}\mathbf{B} \right) &= \rho \mathbf{g}, \\
\frac{\partial \mathbf{B}}{\partial t} + \nabla \cdot (\mathbf{v}\mathbf{B} - \mathbf{B}\mathbf{v}) &= 0, \\
\frac{\partial (\rho e_{\mathrm{tot}})}{\partial t} + \nabla \cdot \left(\left(\rho e_{\mathrm{tot}} + P + \frac{\mathbf{B} \cdot \mathbf{B}}{2} \right) \mathbf{v} - (\mathbf{v} \cdot \mathbf{B}) \mathbf{B} + \mathbf{F}_{\mathrm{rad}} \right) &= 0,
\end{aligned}$$

with $e_{\mathrm{tot}} = e_{\mathrm{i}} + \rho \frac{\mathbf{v} \cdot \mathbf{v}}{2} + \frac{\mathbf{B} \cdot \mathbf{B}}{2} + \rho \Phi$ and $\mathbf{F}_{\mathrm{rad}} \equiv \int_{\nu} \int_{\partial \mathcal{S}^2} I_{\nu}(\Omega) \hat{\mathbf{n}} \, \mathrm{d}\Omega \mathrm{d}\nu$, and the radiative transfer equation :

$$\frac{1}{\rho \kappa_{\nu}} (\hat{\mathbf{n}} \cdot \boldsymbol{\nabla}) I_{\nu} = S_{\nu} - I_{\nu}.$$

Moreover, $\kappa_{\nu} = \kappa_{\nu}(P, T)$ is a function of pressure and temperature, and an equation of state expresses any thermodynamic quantity as a function of two arbitrarily chosen variables.

Operational variables in rMHD simulations

We note that:

- The equation of state relates thermodynamic quantities among each other
- There are different possible choices for a minimal set of operational variables
- Each choice presents advantages and drawbacks.

A possible choice is:

- Density ρ
- Internal energy e_{i}
- Velocity $\mathbf{v}$
- Magnetic field $\mathbf{B}$

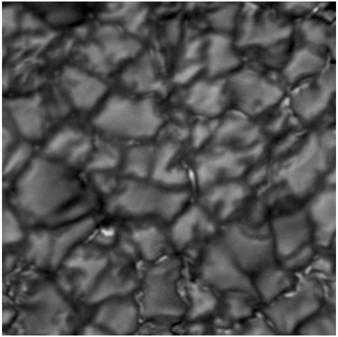
Nota bene:

Any other quantity, such as temperature T, pressure P or optical depth τ, can be derived from this small set of variables, provided an equation of state (EOS) and the relevant opacity data (OPA)!

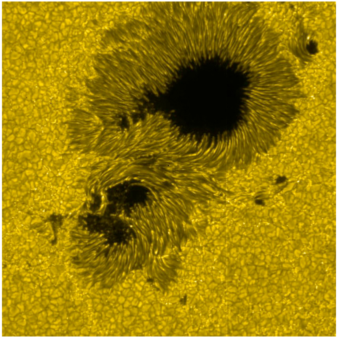
Physical size of simulation data

What is the physical size of a simulation of the solar photosphere?

- Diameter of a small inter-granular bright point: 100km
- Diameter of a convective cell (granule): 1.5Mm
- Diameter of a “typical” sunspot: 100Mm
- Vertical extension from the top of the convection zone to the lower chromosphere: 3Mm



Solar granulation as seen with the Gregor telescope



Sunspot as seen by the SOT on board of the Hinode satellite

Digital size of simulation data

Quick warm-up exercise:

If we represent numbers in single precision (4 bytes), how many GB does it take to store:

- A simulation able to properly resolve inter-granular features such as small bright points, and encompassing $30-40$ granules?
- A simulation able to just resolve inter-granular features such as small bright points, and containing one sunspot in it?

Question:

How much memory do you have on your laptop?

1-byte representation of data: binary to ASCII

00000000	00000001	00000010	00000011	00000100	00000101	00000110	0000011	....
00001000	00001001	00001010	00001011	00001100	00001101	00001110	0000111	....
00010000	00010001	00010010	00010011	00010100	00010101	00010110	0001011	....
00011000	00011001	00011010	00011011	00011100	00011101	00011110	0001111	....
00100000	00100001	00100010	00100011	00100100	00100101	00100110	0010011	!"#$%&’
00101000	00101001	00101010	00101011	00101100	00101101	00101110	0010111	()*+,-./
00110000	00110001	00110010	00110011	00110100	00110101	00110110	0011011	01234567
00111000	00111001	00111010	00111011	00111100	00111101	00111110	0011111	89:;<=>?
01000000	01000001	01000010	01000011	01000100	01000101	01000110	0100011	@ABCDEFG
01001000	01001001	01001010	01001011	01001100	01001101	01001110	0100111	HIJKLMNO
01010000	01010001	01010010	01010011	01010100	01010101	01010110	0101011	PQRSTUVW
01011000	01011001	01011010	01011011	01011100	01011101	01011110	0101111	XYZ[\]^_
01100000	01100001	01100010	01100011	01100100	01100101	01100110	0110011	‘abcdefg
01101000	01101001	01101010	01101011	01101100	01101101	01101110	0110111	hijklmno
01110000	01110001	01110010	01110011	01110100	01110101	01110110	0111011	pqrstuvw
01111000	01111001	01111010	01111011	01111100	01111101	01111110	0111111	xyz{|}~.
10000000	10000001	10000010	10000011	10000100	10000101	10000110	1000011	....
10001000	10001001	10001010	10001011	10001100	10001101	10001110	1000111	....
10010000	10010001	10010010	10010011	10010100	10010101	10010110	1001011	....
10011000	10011001	10011010	10011011	10011100	10011101	10011110	1001111	....
10100000	10100001	10100010	10100011	10100100	10100101	10100110	1010011	....
10101000	10101001	10101010	10101011	10101100	10101101	10101110	1010111	....
10110000	10110001	10110010	10110011	10110100	10110101	10110110	1011011	....
10111000	10111001	10111010	10111011	10111100	10111101	10111110	1011111	....
11000000	11000001	11000010	11000011	11000100	11000101	11000110	1100011	....
11001000	11001001	11001010	11001011	11001100	11001101	11001110	1100111	....
11010000	11010001	11010010	11010011	11010100	11010101	11010110	1101011	....
11011000	11011001	11011010	11011011	11011100	11011101	11011110	1101111	....
11100000	11100001	11100010	11100011	11100100	11100101	11100110	1110011	....
11101000	11101001	11101010	11101011	11101100	11101101	11101110	1110111	....
11110000	11110001	11110010	11110011	11110100	11110101	11110110	1111011	....
11111000	11111001	11111010	11111011	11111100	11111101	11111110	1111111	....

Variable size representation of data: UTF

Extended representations of data

Extended (and fancier) representations of data allows to represent additional symbols by using multiple bytes: UTF-8 and UTF-16 are the most common examples

- ASCII character are still represented by the very same ASCII bytes
- On that specific subclass, ASCI, UTF-8 and UTF-16 are the same
- Loading UTF data in UTF unaware editors might give unexpected results, for instance the UTF symbol "∞" might appear as "â ˆž".

Decimal representation of numbers as text

- The digits 0-9 have their own ASCII representation
- A given decimal number, say 3.141592653e+00, can be therefore represented using the ASCII translation table

This is how π would look like:

00110011	00101110	00110001	00110100	00110001	00110101
3	.	1	4	1	5
00111001	00110010	00110110	00110101	00110011	01100101
9	2	6	5	3	e
00101011	00110000	00110000			
+	0	0			

Extended representations of data

How efficient is this representation in terms of memory? How to do arithmetics?

Hands-on: reading a 1D FALC solar atmosphere

Hands-on

Plot the temperature structure as a function of depth for the FALC 1D model of the solar atmosphere

Hints:

- You are very welcome to use python
- You are very welcome to use python through a Jupyter notebook
- You are very welcome to use the `matplotlib` module as well as the `numpy` module

The IEEE 754 formats

The `float32` and `float64` are now the standard format to represent floating point numbers. `float32` is a 32bit representation:

$$\mathtt{s}\,\overbrace{\mathtt{eeeeeee\ e}}^{\text{exponent}}\overbrace{\mathtt{fffffff\ ffffffff\ ffffffff}}^{\text{coefficient}} \quad (4 \text{ bytes} = 32 \text{ bits})$$

From the point of view of this representation, $\pi = 3.1415927$. In base $b = 2$, we have:

$$\begin{aligned}\pi &= (-1)^{0} \times 1.1001001\ 00001111\ 11011011 \times 10^{1000000\,0 - 0111111\,1} \\ &= (-1)^{s} \times 1.\mathit{fffffff\ ffffffff\ ffffffff} \times 10^{\mathit{eeeeeee\ e} - 0111111\,1)} \\ &= 0\,1000000\ 0\,1001001\ 00001111\ 11011011\end{aligned}$$

Endianness

But wait a minute! `x86` CPUs and their associated memory keep bytes in reverse order!

- *Big-endian* ordering places the most significant byte first, this is the typical ordering for many networking protocols
- *Little-endian* ordering places the least significant byte first

Hands-on:

- Use python to encode π in a 32 bits float and write it to a file
- Use `less` to see the contents of the file
- Use `xxd` to get a binary representation of the file
- Use python to decode π in binary
- What is the `float32` hexadecimal representation of π?

What is your laptop's CPU and memory endianness?

Binary representation of π

The `float32` binary representation of π can be found in python with:

```
''.join([bin(ord(b)).lstrip('0b').rjust(8,'0')+' ' for b in
struct.pack('f',3.1415927)]).rstrip()
```

- `struct.pack('f',3.1415927)` provides you with a string of "raw data" (Python 2) or a bytes string (Python 3) representing π in `float32` format
- For each byte `b` in the string, ord will give you the base 10 representation of that byte (only with Python 2, remove it with Python 3)
- bin will take an integer and represent in bynary form, prefixing it with `0b`
- The rest of the code removes the `0b` prefixes, concatenates the bytes, and displays the whole in a fancy way

Raw binary: create your own format!

- Raw binary *IS NOT* a file format
- Raw binary only means “there is more than just ASCII or UTF”
- It usually also means that numbers are encoded according to the native IEEE 754 formats of your machine

Hands-on

Write a python code that writes and reads $n \times m$ matrices, without using `numpy`, but only the `struct` module

Blind exercise: the FITS file format

Hands-on

Pick any FITS file and try to understand what is inside

Hint: you are welcome to use any shell command and python.

The FITS file format

From wikipedia:

FITS is the most commonly used digital file format in astronomy.
A major feature of the FITS format is that image metadata is stored in a human-readable ASCII header, so that an interested user can examine the headers to investigate a file of unknown origin.

From the FITS support office at NASA:

- *Stands for 'Flexible Image Transport System'*
- *Endorsed by NASA and the International Astronomical Union*
- *Used for the transport, analysis, and archival storage of scientific data sets*
- In particular, used when dealing with *multi-dimensional arrays: 1D spectra, 2D images, 3D+ data cubes*

Fancy file formats for dealing with “big data”

- Simulations run in parallel on supercomputers and involving huge amounts of data might have important I/O overhead
- With distributed storage facilities in supercomputing centres, single files might be written in parallel by different nodes on different physical disks (!)
- In these situations, efficient file formats require extra thinking during their development. . .

HDF5 and netCDF are two extended such examples. However:

- HDF5 is not properly speaking a file format, but a “container”
- These file formats are usually troublesome when used for small-scale applications
- They are however often supported out-of-the-box by visualization programs!

Conclusions about file formats

Question:

What are your conclusions?

Conclusions about file formats

Question:

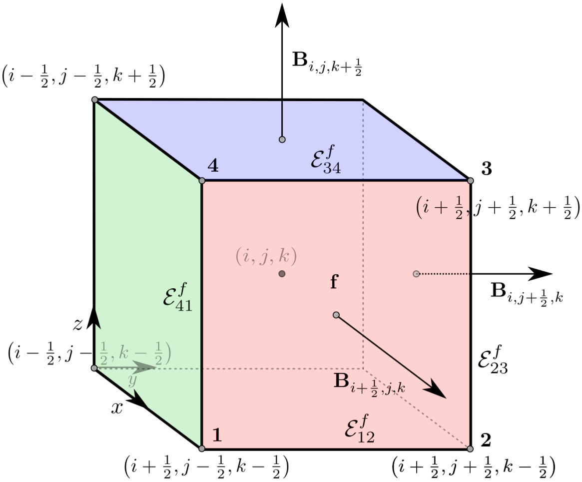
What are your conclusions?

- File formats must be carefully chosen
- There is no universal good choice
- Using one's own format implies implementing all I/O routines and interfacing the required visualization tools
- When it comes to file formats, there are only bad choices, but some of them are extremely bad

Data from CO^{5}BOLD simulations

- CO^{5}BOLD is the COnservative COde for the COmputation of COmpressible COnvection in a BOx of L Dimension with l=2,3
- It stores data using the Universal Input Output (UIO) file format, that is everything except universal
- It stores a minimal amount of data (density, internal energy, velocities and magnetic field)
- Magnetic field is a vector field for which only the perpendicular components to cell boundaries are stored, at cell-centres

Data from CO^{5}BOLD simulations



Cell edges and faces indexing and face-centre averaged magnetic field

Reading UIO datasets

Hands-on

Explore an UIO full file. . .

Data visualization from UIO datasets

Hands-on

Convert a CO^{5}BOLD box into the VDF format and visualize it with VAPOR in 3D…visualize also quantities of your choice in 2D with matplotlib