

ToFu

An open-source python/cython library for synthetic
tomography diagnostics on tokamaks

Laura S. Mendoza¹, Didier Vezinet²

Euroscipy 2019, Bilbao, España

¹INRIA Grand-Est, TONUS Team, Strasbourg, France

²CEA, Cadarache, France

Table of contents

1. Context
2. Tomography diagnostics
3. The ToFu code
4. Demo
5. Optimization of the code
6. What's next

Context

Context: Energy generation



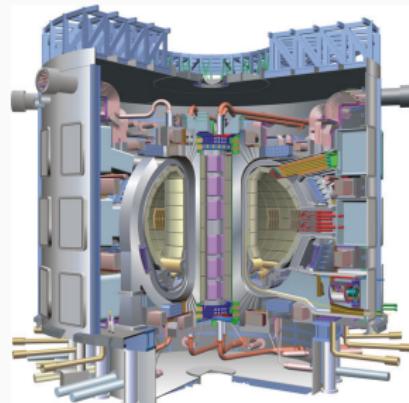
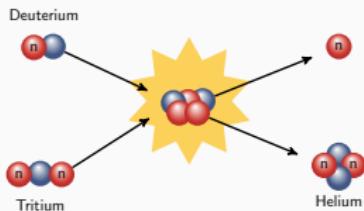
Current solutions present some drawbacks:

- Limited resources
- Production of carbon dioxide
- Radioactive waste
- Not too efficient
- Harmful to surrounding environment

⇒ **Fusion**: cleaner, more reliable, more powerful energy source?

Context: Controlled fusion and magnetic confinement

D-T Fusion reaction



- Gas > 100 Million°K composed of positive ions and negative electrons: plasma

- Confinement using electromagnetic fields

- Energy break-even point still not obtained:

$$Q_{\max} = \frac{E_{\text{output}}}{E_{\text{input}}} = 0.67$$

- Current reactors: different shapes, sizes, heating methods, confinement techniques, etc.

⇒ **Fusion codes:** complexity due to the number of parameters, geometry, model, etc.

Tomography diagnostics

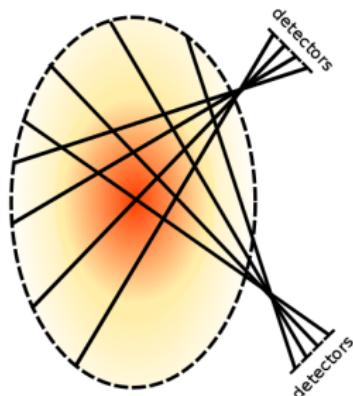
Diagnostics

Set of instruments to measure for the understanding, control and optimization of the plasma performance.

- **Magnetic** diagnostics: currents, plasma stored energy, plasma shape and position;
- **Neutron** diagnostics (ie. cameras, spectrometers, etc.): fusion power;
- Optical systems (**interferometers**): temperature and density profiles;
- Bolometric systems (**tomography**): spatial distribution of radiated power;
- **Spectroscopic**: X-ray wavelength range, impurity species and density, input particle flux, ion temperature, helium density, fueling ratio, plasma rotation, and current density.

Tomography diagnostics - numerical context

$$M_i(t) = \iiint_{V_i} \overrightarrow{\varepsilon(x,t)} \cdot \vec{n} \Omega_i \, dV$$



- **Direct problem** (synthetic diagnostic):

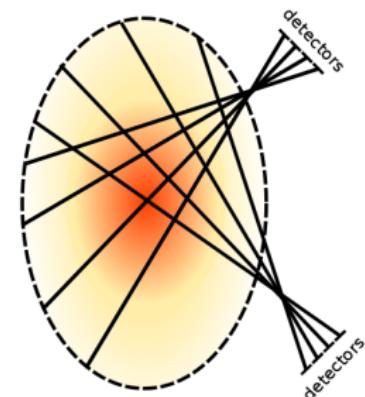
Simulated emissivity \longrightarrow integrated measurements

- **Inverse problem** (tomography):

Integrated measurements \longrightarrow Reconstructed emissivity

Tomography diagnostics - numerical context

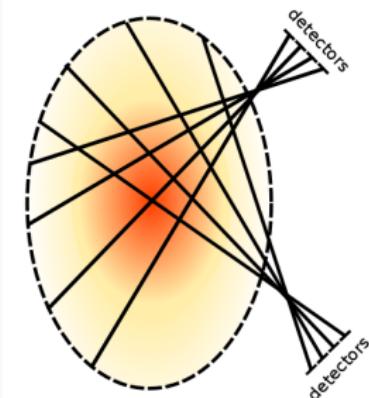
$$M_i(t) = \iiint_{V_i} \overrightarrow{\varepsilon(x, t)} \cdot \vec{n} \Omega_i \, dV$$



- **Direct problem** (synthetic diagnostic):
Simulated emissivity → measurements
Spatial integration
- **Inverse problem** (tomography):
Integrated measurements → Reconstructed emissivity
Mesh and basis functions construction, spatial integration, data filtering, inversion routines, etc.

Tomography diagnostics - numerical context

$$M_i(t) = \iiint_{V_i} \overrightarrow{\varepsilon(x,t)} \cdot \vec{n} \Omega_i \, dV$$



- **Direct problem** (synthetic diagnostic):
Simulated emissivity → measurements
Spatial integration
- **Inverse problem** (tomography):
Integrated measurements → Reconstructed emissivity
Mesh and basis functions construction, spatial integration, data filtering, inversion routines, etc.

Tomography very sensitive to errors, noise and bias
→ Reputation for low reproducibility / reliability

The ToFu code

Motivation: “current” state

In the fusion community, codes for synthetic diagnostic are developed:

- by physicists (with little to no programming experience),
- in Matlab,
- from scratch,
- in local/private

... which means

- repetition of work: lost time, man-power, etc;
- no traceability,
- results impossible to reproduce,
- no standardization of diagnostics

A code for Tomography for Fusion

Develop a common tool:

- Accessible to everyone (open-source)
- Generic (geometry independent)
- Portable (developed in Python)
- Optimized (reliability and performance)
- Documented online
- Continuous integration



For tomography diagnostics:

The **Tomography for Fusion** code (**ToFu**¹²³)

¹repository: <https://github.com/ToFuProject/tofu>

²documentation: <https://tofuproject.github.io/tofu/index.html>

³didier2016.

More about Tofu

- Created in 2014
- Open Source: **MIT license**
- Python 2.7 and **Python 3 + Cython**
- Continuous integration: **Travis CI**
- **conda, pip**
- Two (main) developers:
 - ▶ Didier Vezinet (creator, PhD in Physics)
 - ▶ Laura S. Mendoza (since June 2018, PhD in Applied Mathematics)
- Contributors:
 - ▶ Jorge Morales (PhD in Physics)
 - ▶ Florian Le Bourdais
 - ▶ Arpan Khandelwal (intern)



What ToFu can do: modeling of simplified geometry

What ToFu can do: 3D modeling of a 1D camera

What ToFu can do: 3D modeling of a 2D camera

What ToFu can do: handle basic reflexions

What ToFu can do: computing synthetic signals

What ToFu can do

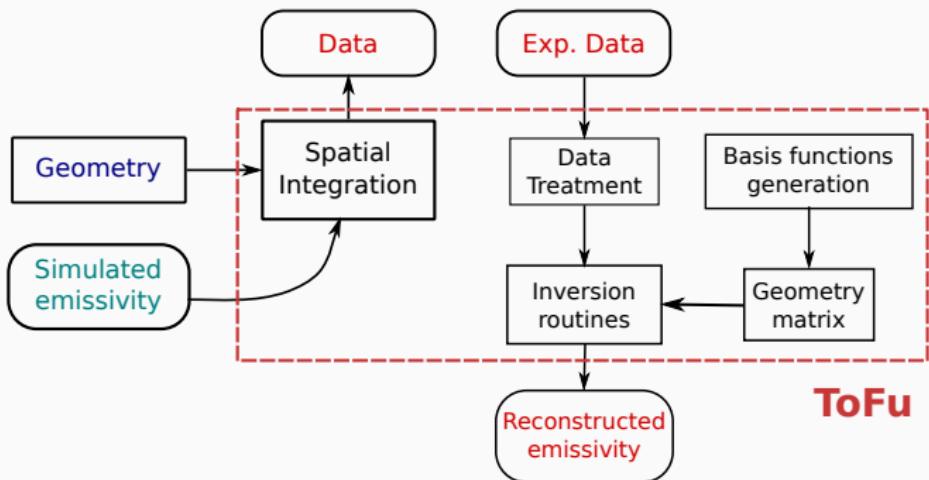
ToFu can:

- Modeling of simplified geometry
- 3D modeling of a 1D camera
- 3D modeling of a 2D camera
- Handle basic reflexions
- Computing synthetic signals
- ...and soon:
 - ▶ meshing and basis functions
 - ▶ tomographic inversion
 - ▶ dust particle trajectory tracking
 - ▶ faster Matplotlib + PyQtGraph visualization
 - ▶ magnetic field line tracing
 - ▶ data visualization and statistical tools (pandas)

Demo

Demo

Tofu's structure



ToFu

Optimization of the code

Geometry reconstruction: ray-tracing techniques

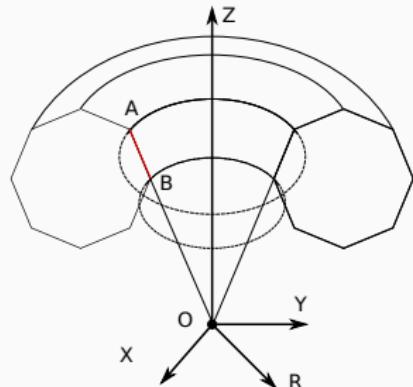
To reconstruct emissivity we need to take account:

- Up to hundreds of structural elements in vessel
 - Scale of the vessel: 10^4 bigger than smaller structural detail
- ⇒ Geometry defined with minimal data polygon (R, Z)
extruded along φ
- ⇒ Symmetry of vessel along φ



Optimization of ray-tracing algorithm

- Description of geometry:
 - ▶ Vessel and structures: set of 2D polygon
$$\mathcal{P}_j = \bigcup_{i=1}^n \overline{A_i B_i}$$
 - ▶ Extruded along $[\varphi_{min}, \varphi_{max}]$
 - ▶ Detectors defined as set of rays (of origin D and direction u)
⇒ Light memory-wise
- ⇒ Equivalent to: set of truncated cones
(frustums) of generatrix $A_i B_i$



Ray-tracing algorithm on fusion device → Computation of cone-Ray intersection

$$\exists (q, k) \in [0; 1] \times [0; \infty[, \quad \left\{ \begin{array}{l} R - R_A = q(R_B - R_A) \\ Z - Z_A = q(Z_B - Z_A) \\ DM = ku \end{array} \right.$$

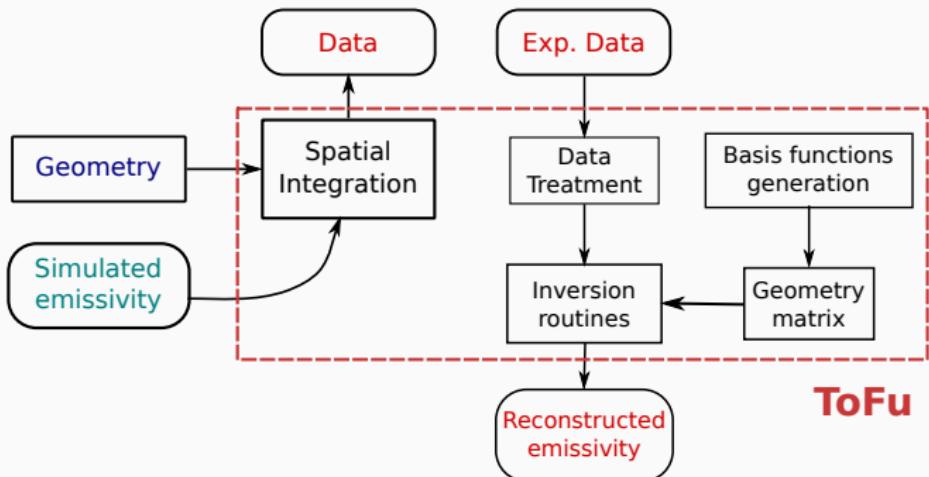
Optimization of ray-tracing algorithm

Cone-Ray intersection algorithm:

- Main steps:
 - ▶ Test if intersection with bounding-box
 - ▶ Computation of special cases of segment AB
 - ▶ Computation of special cases of ray directional vector
 - ▶ General case: solution of a quadratic equation
- Pre-computation of geometry-independent variables
- Core functions written in **Cython**
- Parallelization over ray-loop (**prange** loops)

Nb LOS	10^3	10^4	10^5	10^6	
original	$3.26 \cdot 10^1$	$3.10 \cdot 10^2$	$3.20 \cdot 10^3$	$3.17 \cdot 10^4$	(8h48)
optimized	$2.58 \cdot 10^{-2}$	$2.72 \cdot 10^{-1}$	2.74	$2.66 \cdot 10^1$	(< 30s)
32 threads	$1.36 \cdot 10^{-2}$	$4.66 \cdot 10^{-2}$	$3.64 \cdot 10^{-1}$	2.92	

Tofu's structure



Optimization of spatial integration routines

For the integration along a LOS:

- Sample (discretization) of a 3D ray:
 - ▶ Different quadrature rules: midpoint, simpson, romberg
 - ▶ Resolution given by user
 - ▶ Possible to define a sub-domain of discretization
- Integration of a python function **func** defined by user by:
 - ▶ **numpy.sum**
 - ▶ Cython based sum
 - ▶ **Scipy.integrate.simps**
 - ▶ **Scipy.integrate.romb**
- Optional optimizations:
 - ▶ calls to **func**: avoid Cython-Python conversion, user-defined
 - ▶ memory: fine resolutions, high number of LOS
 - ▶ hybrid: compromise

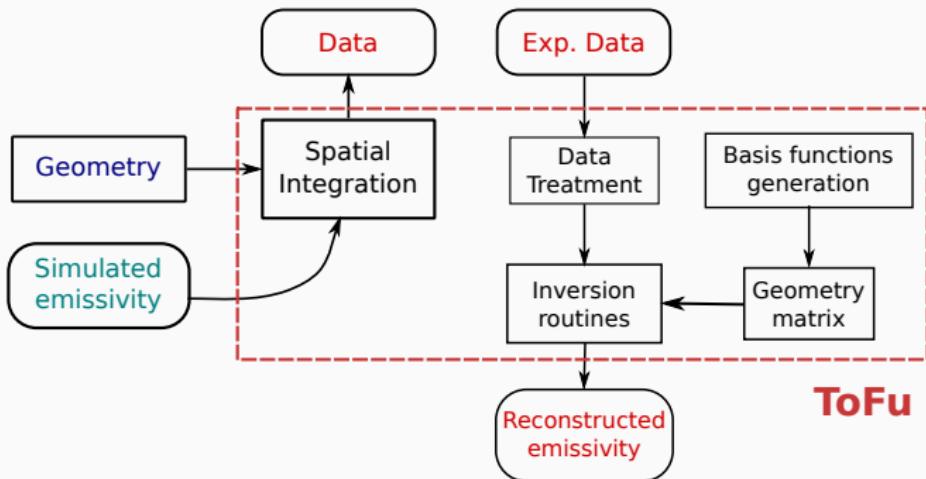
Optimization of spatial integration routines

LOS	10	10^2	10^3	10^4
original	0.46	2.24	18.1	x
memory	0.9	8.9	96	945 (6Gb)
calls	0.207	0.53	4.32	x
hybrid	0.08	0.44	4.2	40.3 (32Gb)

- Space resolution: 10^{-3}
- Number of time steps: 10^3
- Integration method: **sum** (Cython or numpy) on midpoint

What's next

Tofu's structure



ToFu

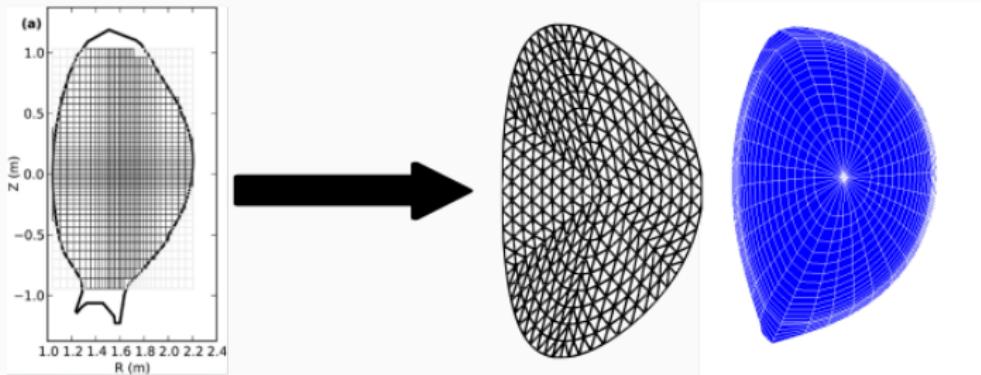
On geometry discretization: meshing

Several options for poloidal cut meshing:

- Cartesian mesh
- Polar mesh
- Adaptive polar mesh
- Hexagonal mesh
- Triangular mesh

For basis functions:

- Lagrange polynomials
- B-splines
- NURBS
- Box-splines



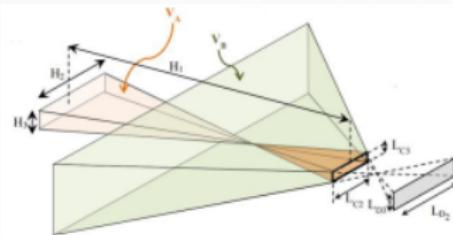
Tofu's main algorithms

Visualization and geometrical tools:

- Ray-tracing algorithm
- Distance 3D objects and rays
- Vignetting: 3D polygon - Ray
- Solid angles computation (reflexions)

For direct and inverse problem:

- Basis functions
- Discretization: Geometry, Lines, Volumes
- Spatial integration
- Regularization-inversion routines (Bayesian, non linear, etc.)
- Filtering



Thank you for your attention!

B(asis)-Splines basis*

B-Splines of degree d are defined by the **recursion formula**:

$$B_j^{d+1}(x) = \frac{x - x_j}{x_{j+d} - x_j} B_j^d(x) + \frac{x_{j+1} - x}{x_{j+d+1} - x_{j+1}} B_{j+1}^d(x) \quad (1)$$

Some important properties about B-splines:

- Piece-wise polynomials of degree $d \Rightarrow$ **smoothness**
- Compact support \Rightarrow **sparse matrix system**
- Partition of unity $\sum_j B_j(x) = 1, \forall x \Rightarrow$ **conservation laws**

