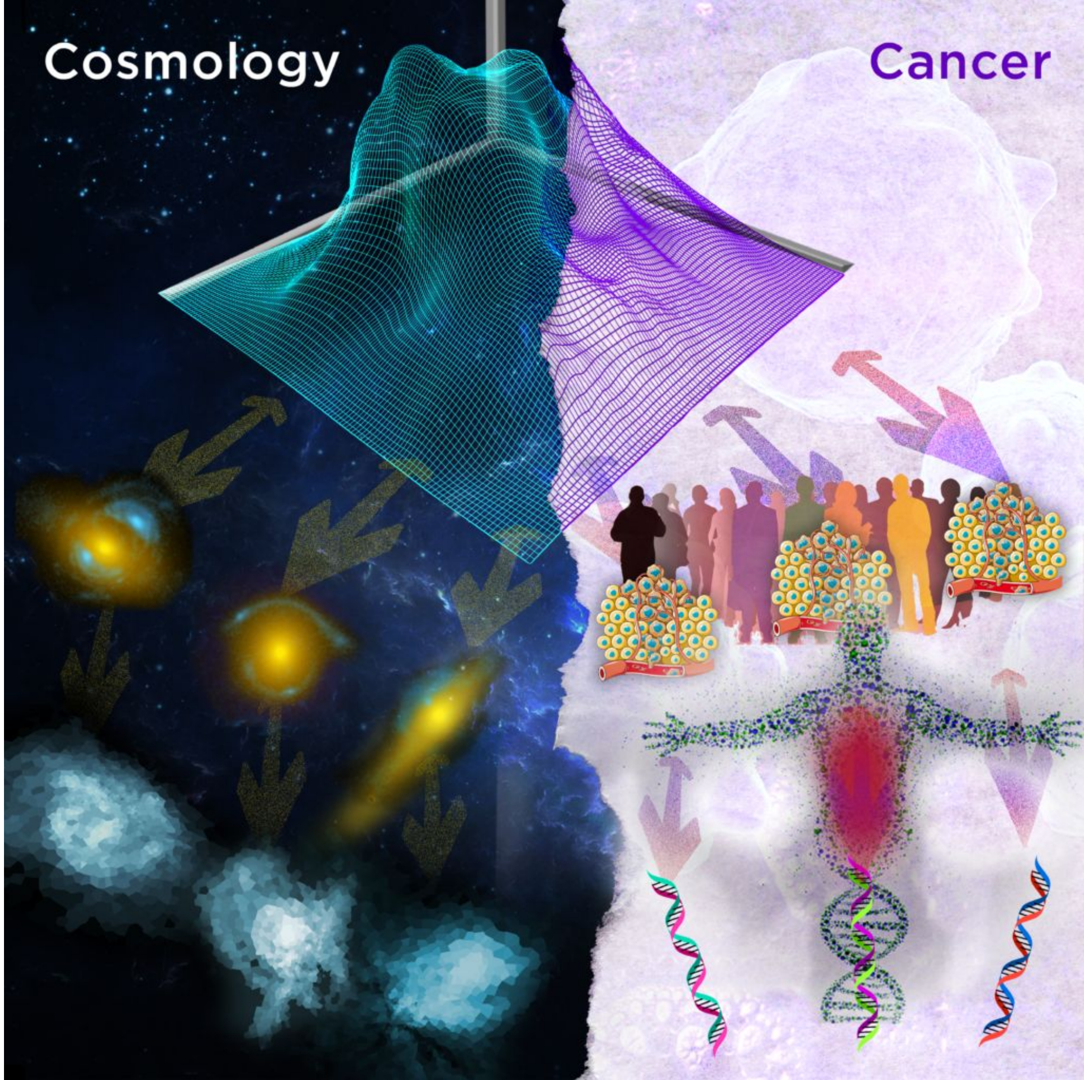
PyAutoFit: Classy Probabilistic Programming for Data Science

James Nightingale

Richard Hayes, Matthew Griffiths

www.jamesnightingale.net

Durham University



Overview

PyAutoFit & Probabilistic Programming:

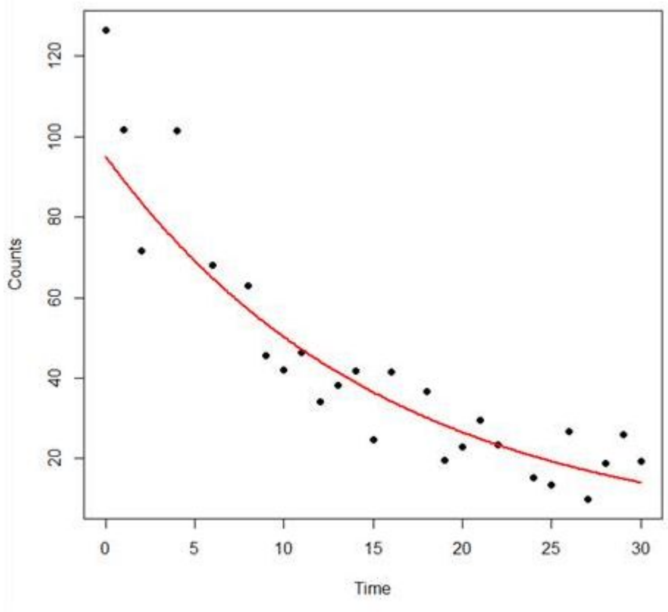
- What is probabilistic programming?
- What is PyAutoFit?
- Why PyAutoFit?

Cosmology:

- Description of example use-case - strong gravitational lensing.
- Application to Astronomy data.
- Building multi-level models via Python classes.

Model Fitting

Given some data and a model, finding the set of model parameters that provide the best fit to the data.



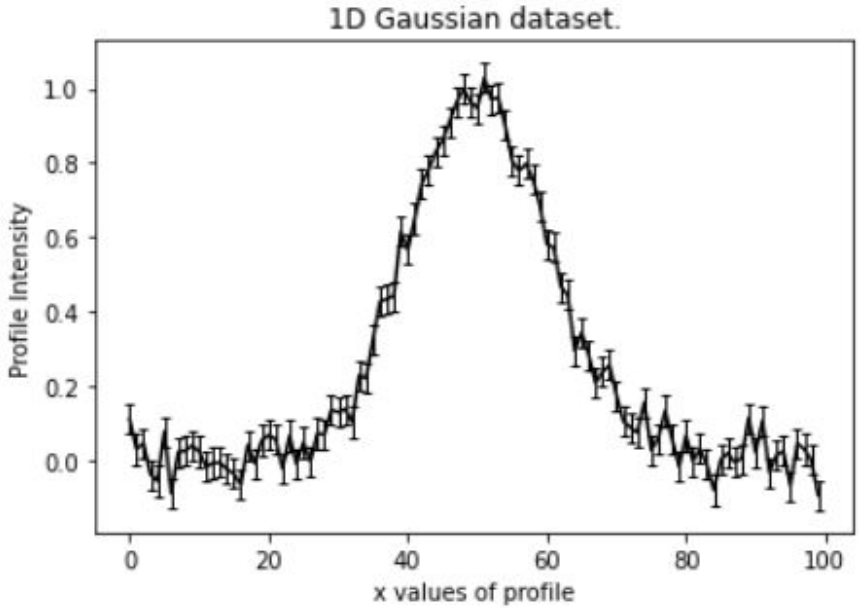
$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B)}$$

Model Fitting

Model -> Gaussian:

- Centre
- Intensity
- Sigma

1) Draw a set of parameters.
2) Create Model Gaussian.
3) Fit to Dataset.
4) Compute Likelihood.
5) Repeat using non-linear search.

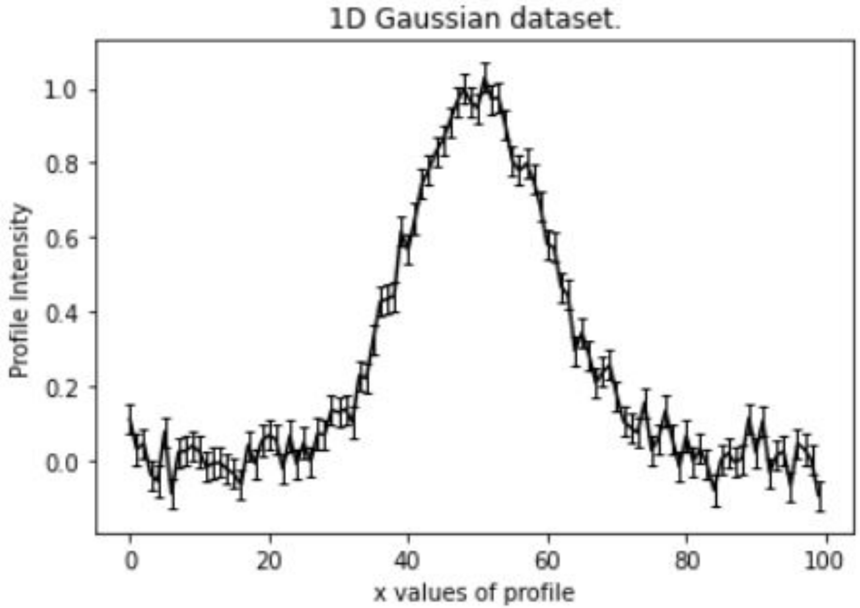


Model Fitting

Model -> Gaussian:

- Centre **= 60.0**
- Intensity **= 20.0**
- Sigma **= 15.0**

1) **Draw a set of parameters.**
2) Create Model Gaussian.
3) Fit to Dataset.
4) Compute Likelihood.
5) Repeat using non-linear search.

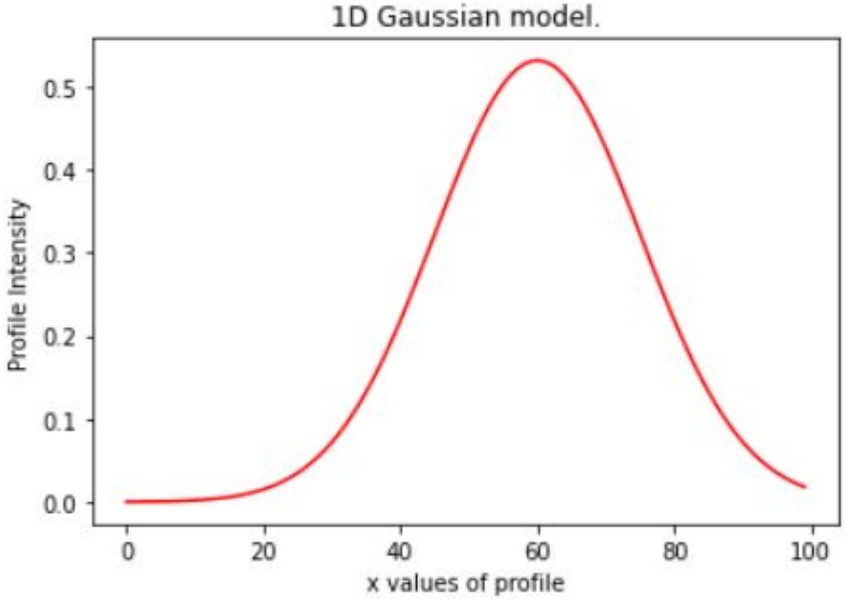


Model Fitting

Model -> Gaussian:

- Centre **= 60.0**
- Intensity **= 20.0**
- Sigma **= 15.0**

1) Draw a set of parameters.
2) **Create Model Gaussian.**
3) Fit to Dataset.
4) Compute Likelihood.
5) Repeat using non-linear search.

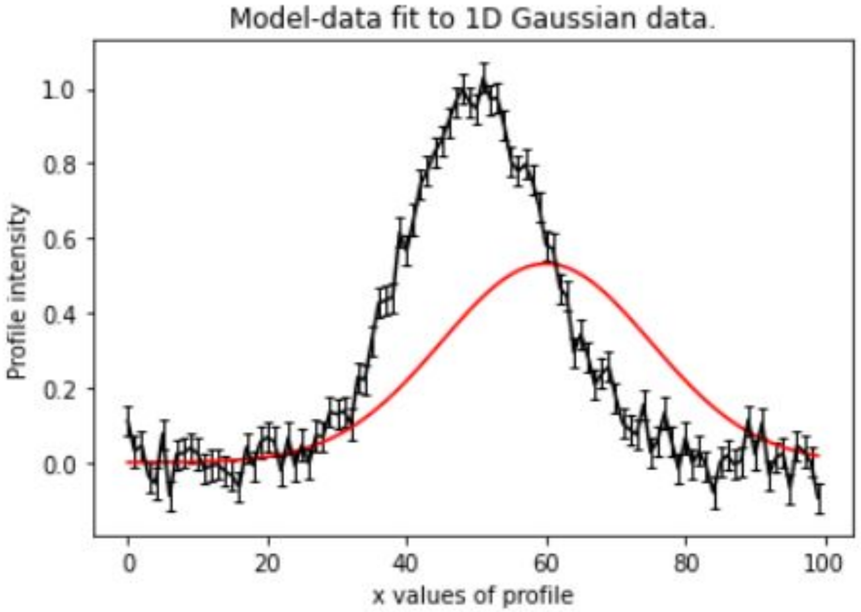


Model Fitting

Model -> Gaussian:

- Centre **= 60.0**
- Intensity **= 20.0**
- Sigma **= 15.0**

1) Draw a set of parameters.
2) Create Model Gaussian.
3) **Fit to Dataset.**
4) Compute Likelihood.
5) Repeat using non-linear search.

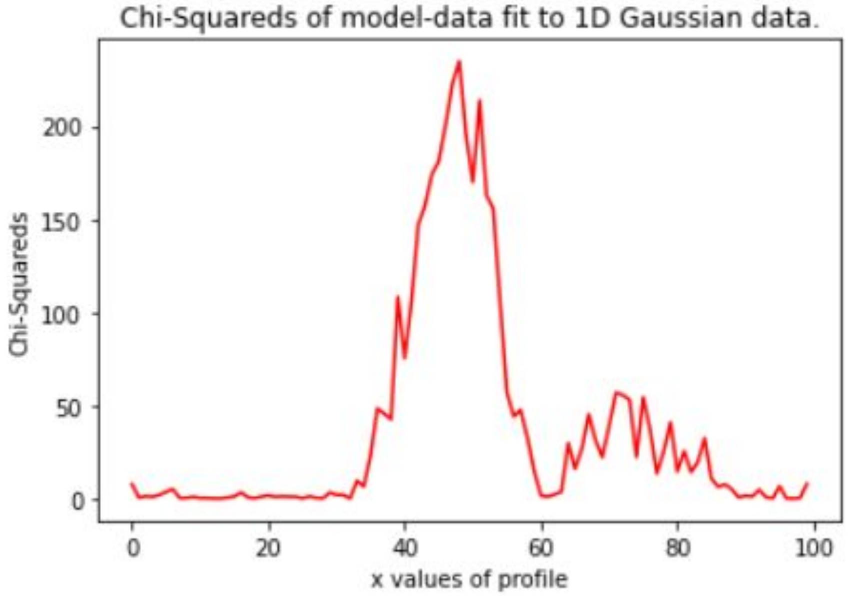


Model Fitting

Model -> Gaussian:

- Centre **= 60.0**
- Intensity **= 20.0**
- Sigma **= 15.0**

1) Draw a set of parameters.
2) Create Model Gaussian.
3) Fit to Dataset.
4) **Compute Likelihood.**
5) Repeat using non-linear search.

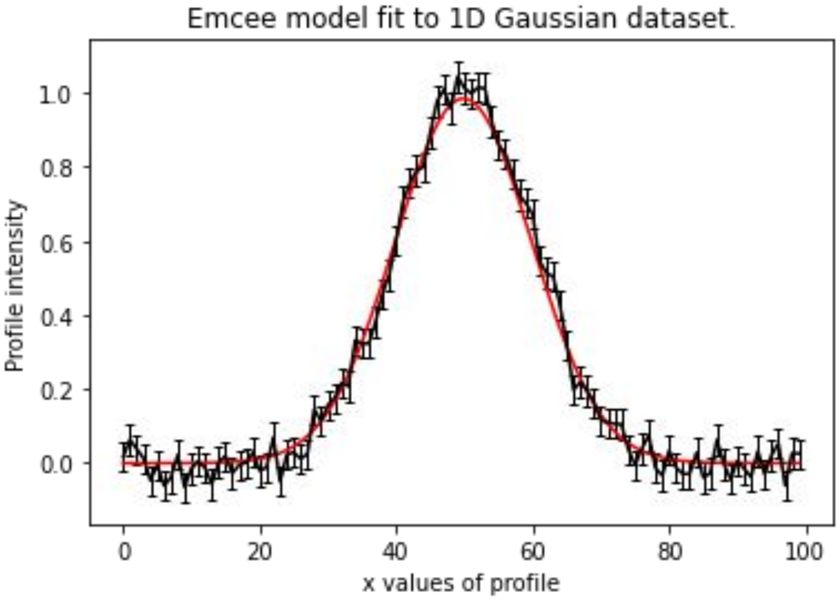


Model Fitting

Model -> Gaussian:

- Centre **= 60.0**
- Intensity **= 20.0**
- Sigma **= 15.0**

1) Draw a set of parameters.
2) Create Model Gaussian.
3) Fit to Dataset.
4) Compute Likelihood.
5) **Repeat using non-linear search.**



Probabilistic Programming

What is Probabilistic Programming?

Probabilistic programming languages (PPL) provide a framework that allows users to easily specify a probabilistic model and perform inference automatically.

- There are a plethora of PPL's available (e.g. **PyMC3, STAN, Pyro**).
- All are suited to different problems, have different core features, etc.

They are some of the Github mega projects, so why on Earth are we developing our own PPL?

```
import pymc3 as pm

X, y = linear_training_data()
with pm.Model() as linear_model:
    weights = pm.Normal("weights", mu=0, sigma=1)
    noise = pm.Gamma("noise", alpha=2, beta=1)
    y_observed = pm.Normal(
        "y_observed",
        mu=X @ weights,
        sigma=noise,
        observed=y,
    )

    prior = pm.sample_prior_predictive()
    posterior = pm.sample()
    posterior_pred = pm.sample_posterior_predictive(posterior)
```

PyAutoFit: A PPL for Astronomers (and data science!)

Existing PPL's not suited to the model fitting challenges we faced when in Astronomy, for example:

PyAutoFit: A PPL for Astronomers (and data science!)

Existing PPL's not suited to the model fitting challenges we faced when in Astronomy, for example:

- Fitting large and homogenous datasets with an identical model fitting procedure, with tools for processing the large libraries of results output.

PyAutoFit: A PPL for Astronomers (and data science!)

Existing PPL's not suited to the model fitting challenges we faced when in Astronomy, for example:

- Fitting large and homogenous datasets with an identical model fitting procedure, with tools for processing the large libraries of results output.
- Problems where likelihood evaluations are expensive (e.g. run times of days per model-fit), necessitating highly customizable model-fitting pipelines with support for massively parallel computing.

PyAutoFit: A PPL for Astronomers (and data science!)

Existing PPL's not suited to the model fitting challenges we faced when in Astronomy, for example:

- Fitting large and homogenous datasets with an identical model fitting procedure, with tools for processing the large libraries of results output.
- Problems where likelihood evaluations are expensive (e.g. run times of days per model-fit), necessitating highly customizable model-fitting pipelines with support for massively parallel computing.
- Fitting many different models to the same dataset with tools that streamline model comparison.

PyAutoFit: A PPL for Astronomers (and data science!)

Existing PPL's not suited to the model fitting challenges we faced when in Astronomy, for example:

- Fitting large and homogenous datasets with an identical model fitting procedure, with tools for processing the large libraries of results output.
- Problems where likelihood evaluations are expensive (e.g. run times of days per model-fit), necessitating highly customizable model-fitting pipelines with support for massively parallel computing.
- Fitting many different models to the same dataset with tools that streamline model comparison.

PyAutoFit: highly customizable model-fitting software, for big data challenges in the many model regime.

PyAutoFit: Links / Overview

PyAutoFit

GitHub: https://github.com/rhayes777/PyAutoFit

Readthedocs: https://pyautofit.readthedocs.io/en/latest/

JOSS Paper: https://joss.theoj.org/papers/10.21105/joss.02550

Binder: https://mybinder.org/v2/gh/Jammy2211/autofit_workspace/HEAD

HowToFit

Teach **anyone** how to compose and fit a probabilistic model with **PyAutoFit**.

We can also use it to get a model instance of the `median_pdf` model, which is the model where each parameter is the value estimated from the probability distribution of parameter space.

In [14]:

```
mp_instance = result.samples.median_pdf_instance
print()
print("Median PDF Model:\n")
print("Centre = ", mp_instance.centre)
print("Intensity = ", mp_instance.intensity)
print("Sigma = ", mp_instance.sigma)
```

```
Median PDF Model:

Centre =  49.92285569756167
Intensity =  24.974961843717058
Sigma =  9.969794911012947
```

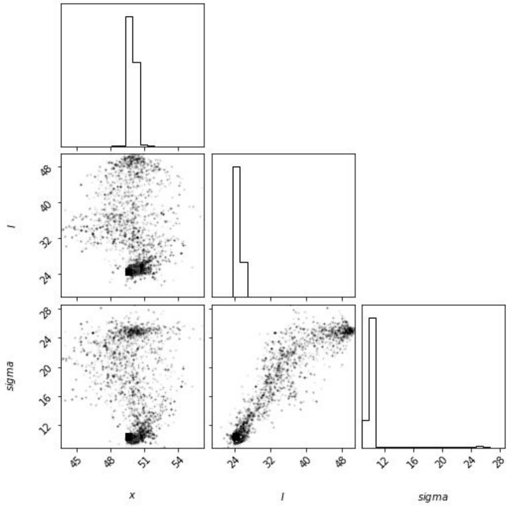
The Probability Density Functions (PDF's) of the results can be plotted using the Emcee's visualization tool `corner.py`, which is wrapped via the `EmceePlotter` object.

The PDF shows the 1D and 2D probabilities estimated for every parameter after the model-fit. The two dimensional figures can show the degeneracies between different parameters, for example how increasing σ and decreasing the intensity I can lead to similar likelihoods and probabilities.

In [15]:

```
emcee_plotter = aplt.EmceePlotter(samples=result.samples)
emcee_plotter.corner()
```

```
2021-07-26 16:42:47,675 - root - WARNING - Too few points to create valid contours
2021-07-26 16:42:47,712 - root - WARNING - Too few points to create valid contours
2021-07-26 16:42:47,737 - root - WARNING - Too few points to create valid contours
```

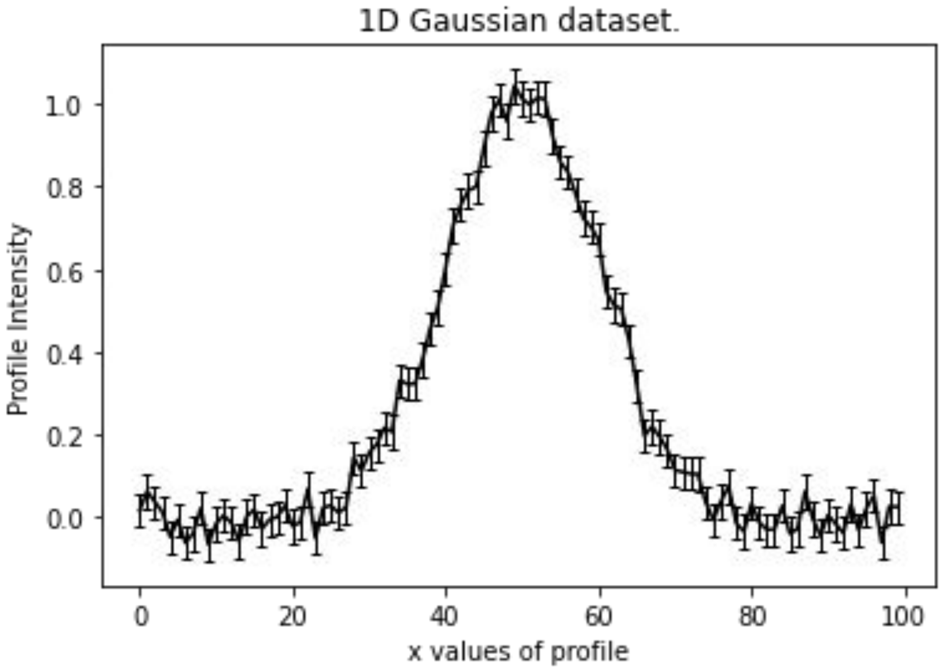


PyAutoFit: Classy Interface

PyAutoFit: Classy Probabilistic Programming

Illustrative example, fitting noisy 1D data of a Gaussian.

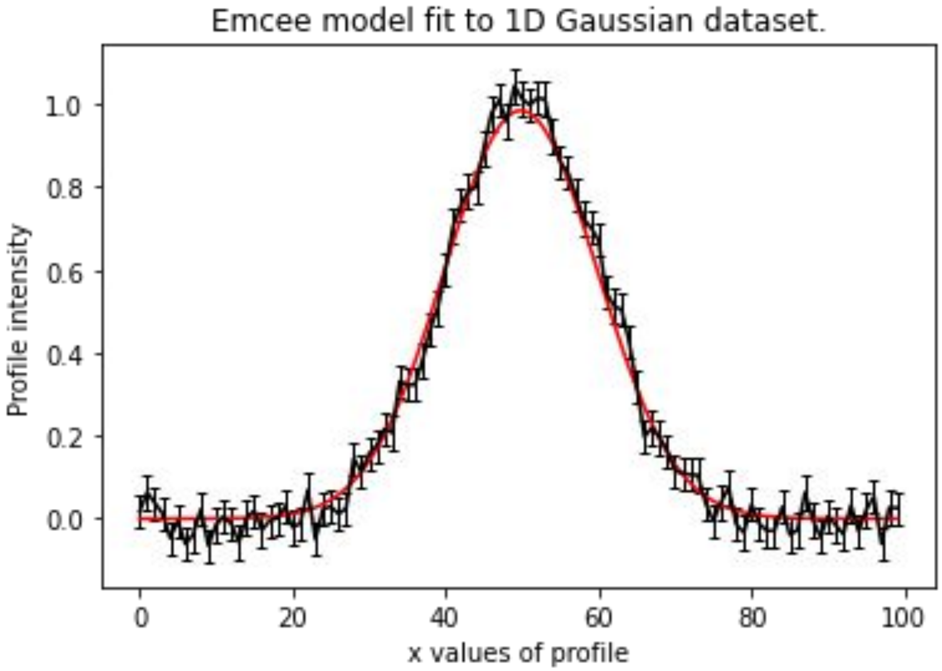
Aim: use **PyAutoFit** to fit a Gaussian to the dataset via a non-linear search.



PyAutoFit: Classy Probabilistic Programming

Illustrative example, fitting noisy 1D data of a Gaussian.

Aim: use **PyAutoFit** to fit a Gaussian to the dataset via a non-linear search.



PyAutoFit: Classy Probabilistic Programming

Illustrative example, fitting noisy 1D data of a Gaussian.

- Write a Python class to define the **model component.**

```
class Gaussian:

    def __init__(
        self,
        centre=0.0,        # <- PyAutoFit recognises these
        intensity=0.1,     # <- constructor arguments are
        sigma=0.01,        # <- the Gaussian's parameters.
    ):
        self.centre = centre
        self.intensity = intensity
        self.sigma = sigma

    """
    An instance of the Gaussian class will be available during model fitting.

    This method will be used to fit the model to ``data`` and compute a likelihood.
    """

    def profile_from_xvalues(self, xvalues):

        transformed_xvalues = xvalues - self.centre

        return (self.intensity / (self.sigma * (2.0 * np.pi) ** 0.5)) * \
                np.exp(-0.5 * transformed_xvalues / self.sigma)
```

PyAutoFit: Classy Probabilistic Programming

Illustrative example, fitting noisy 1D data of a Gaussian.

- Write a Python class to define the **model component.**
- Write an Analysis class with the **data** and **likelihood function.**

```
class Analysis(af.Analysis):

    def __init__(self, data, noise_map):

        self.data = data
        self.noise_map = noise_map

    def log_likelihood_function(self, instance):

        """
        The 'instance' that comes into this method is an instance of the Gaussian class
        above, with the parameters set to values chosen by the non-linear search.
        """

        print("Gaussian Instance:")
        print("Centre = ", instance.centre)
        print("Intensity = ", instance.intensity)
        print("Sigma = ", instance.sigma)

        """
        We fit the ``data`` with the Gaussian instance, using its
        "profile_from_xvalues" function to create the model data.
        """

        xvalues = np.arange(self.data.shape[0])

        model_data = instance.profile_from_xvalues(xvalues=xvalues)
        residual_map = self.data - model_data
        chi_squared_map = (residual_map / self.noise_map) ** 2.0
        log_likelihood = -0.5 * sum(chi_squared_map)

        return log_likelihood
```

PyAutoFit: Classy Probabilistic Programming

Illustrative example, fitting noisy 1D data of a Gaussian.

- Write a Python class to define the **model component.**
- Write an Analysis class with the **data** and **likelihood function.**
- Combine with your favourite **non-linear search** to fit the model to the data.

```
model = af.Model(Gaussian)

analysis = Analysis(data=data, noise_map=noise_map)

emcee = af.Emcee(nwalkers=50, nsteps=2000)

result = emcee.fit(model=model, analysis=analysis)
```

PyAutoFit: Classy Probabilistic Programming

Illustrative example, fitting noisy 1D data of a Gaussian.

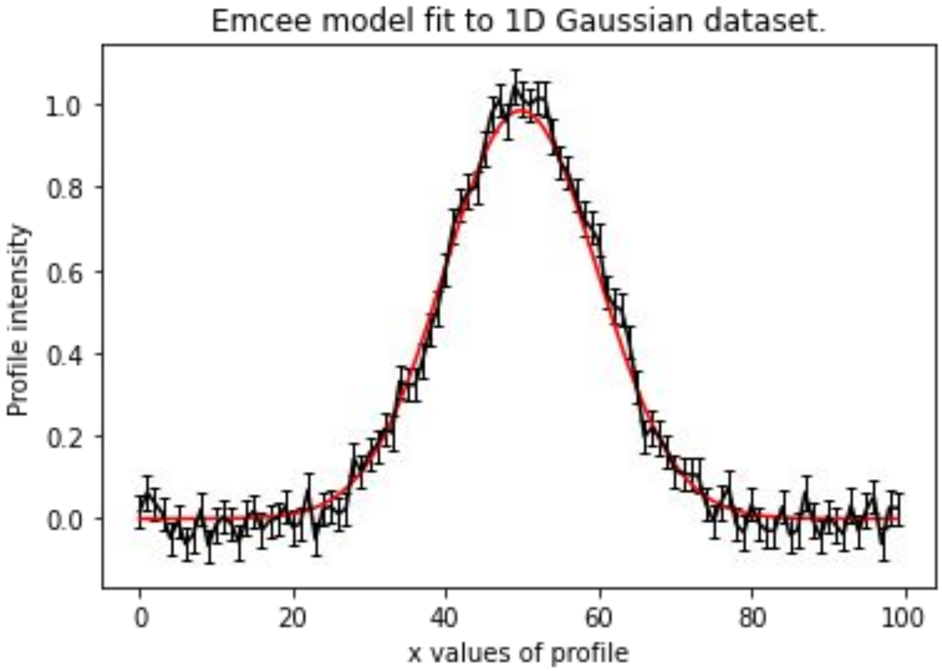
- Write a Python class to define the **model component.**
- Write an Analysis class with the **data** and **likelihood function.**
- Combine with your favourite **non-linear search** to fit the model to the data.

```
model = af.Model(Gaussian)

analysis = Analysis(data=data, noise_map=noise_map)

emcee = af.Emcee(nwalkers=50, nsteps=2000)

result = emcee.fit(model=model, analysis=analysis)
```

PyAutoFit: Classy Probabilistic Programming

Illustrative example, fitting noisy 1D data of a Gaussian.

Aim: use **PyAutoFit** to fit a Gaussian to the dataset via a non-linear search.



PyAutoFit: Classy Probabilistic Programming

Illustrative example, fitting noisy 1D data of a Gaussian.

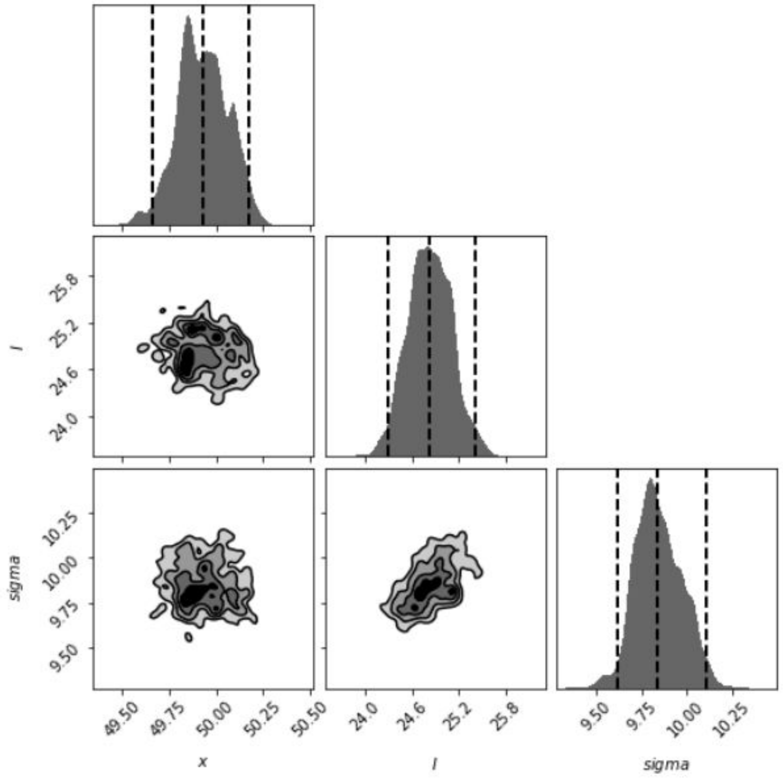
- Write a Python class to define the **model component.**
- Write an Analysis class with the **data** and **likelihood function.**
- Combine with your favourite **non-linear search** to fit the model to the data.
- **Result** object contains all the information you need on your model-fit.

```
samples = result.samples

print("Final 10 Parameters:")
print(samples.parameter_lists[-10:])

print("Sample 10`s third parameter value (Gaussian -> sigma)")
print(samples.parameter_lists[9][2], "\n")

median_pdf_vector = samples.median_pdf_vector

vector_at_upper_sigma = samples.vector_at_upper_sigma(sigma=3.0)
vector_at_lower_sigma = samples.vector_at_lower_sigma(sigma=3.0)

print("Upper Parameter values w/ error (at 3.0 sigma confidence):")
print(vector_at_upper_sigma)
print("lower Parameter values w/ errors (at 3.0 sigma confidence):")
print(vector_at_lower_sigma, "\n")
```

PyAutoFit: Classy Probabilistic Programming

Illustrative example, fitting noisy 1D data of a Gaussian.

- Write a Python class to define the **model component.**
- Write an Analysis class with the **data** and **likelihood function.**
- Combine with your favourite **non-linear search** to fit the model to the data.
- **Result** object contains all the information you need on your model-fit.



PyAutoFit: Customization

Python Classes

The use of Python Classes to define the model, analysis and non-linear searches has downsides relative to other PPLs:

- It is a less concise interface.
- It requires a basic understanding of Python classes and object oriented programming (albeit good documentation can alleviate this).

The benefit is it provides a far more customizable model-fitting experience.

Customizing the Model

Full customization of the model parameterization, priors and valid regions of parameter space.

- Default priors can be specified in easy to set up configuration files, so a new user does not need to ‘think’ about them.

```
"""
Compose model with multiple-components.
"""

gaussian_0 = af.Model(Gaussian)
gaussian_1 = af.Model(Gaussian)

"""
Manually set prior on each parameter.
"""

gaussian_0.centre = af.UniformPrior(lower_limit=0.0, upper_limit=100.0)
gaussian_0.intensity = af.LogUniformPrior(lower_limit=0.0, upper_limit=1e2)
gaussian_0.sigma = af.GaussianPrior(mean=10.0, sigma=5.0, lower_limit=0.0, upper_limit=np.inf)

"""
Fix a parameter to a value (reducing dimensionality of parameter space by 1).
"""
gaussian_0.sigma = 0.5

"""
Link two parameters in a model (reducing dimensionality of parameter space by 1).
"""
gaussian_0.centre = gaussian_1.centre

"""
Make assertions removing regions of parameter space.
"""
gaussian_1.add_assertion(gaussian_1.sigma > 5.0)

"""
To make a model with multiple components we use a `Collection` object.
"""
model = af.Collection(gaussian_0=gaussian_0, gaussian_1=gaussian_1)
```

Customizing the Model

Full customization of the model parameterization, priors and valid regions of parameter space.

- Straightforward to add many different model-components via inheritance.
- **Composition** makes this concise and scalable.

```
class Gaussian:

    def __init__(
            self,
            centre=0.0,
            intensity=0.1,
            sigma=0.01,
    ):
        self.centre = centre
        self.intensity = intensity
        self.sigma = sigma

class GaussianKurtosis(Gaussian):

    def __init__(
            self,
            centre=0.0,
            intensity=0.1,
            sigma=0.01,
            kurtosis=0.1,
    ):

        super().__init__(
            centre=centre,
            intensity=intensity,
            sigma=sigma
        )

        self.kurtosis = kurtosis

class Exponential:
    def __init__(
        self,
        centre=0.0,
        intensity=0.1,
        rate=0.01,
    ):

        self.centre = centre
        self.intensity = intensity
        self.rate = rate
```

Customizing the Analysis

The Analysis class can be extended or provide model-specific on-the-fly visualization of the model-fit so far.

- Uses the maximum likelihood model of the search so far.
- For long model-fits can inform you if the fitting has gone wrong early.

```
class Analysis(af.Analysis):
    def __init__(self, data, noise_map):

        self.data = data
        self.noise_map = noise_map

    def log_likelihood_function(self, instance):

        ...

    def visualize(self, paths, instance):

        """
        During a model-fit, the `visualize` method is called throughout the
        non-linear search. The `instance` is maximum log likelihood solution
        obtained so far and is used to output on-the-fly images.
        """
        xvalues = np.arange(self.data.shape[0])

        model_data = instance.profile_from_xvalues(xvalues=xvalues)
        residual_map = self.data - model_data

        plt.errorbar(
            x=xvalues, y=residual_map, color="k", ecolor="k",
        )
        plt.title("1D Residual Map")
        plt.xlabel("x value of profile")
        plt.ylabel("Residual")
        plt.savefig(path.join(paths.image_path, "residual_map.png"))
        plt.clf()
```

Customizing the Search

PyAutoFit supports many non-linear searches (MCMC, nested sampling, optimizers, etc.).

- Full customization of their settings.
- Defaults to configuration file values if not specified.

```
emcee = af.Emcee(
    name="example_mcmc",
    nwalkers=50,
    nsteps=2000,
    initializer=af.InitializerBall(lower_limit=0.49, upper_limit=0.51),
    auto_correlations_settings=af.AutoCorrelationsSettings(
        check_for_convergence=True,
        check_size=100,
        required_length=50,
        change_threshold=0.01,
    ),
)
```

PyAutoFit: Features

Database

Results of many model fits are output in an sqlite relational database:

- Allocated a **unique identifier** based on the model-fit, such that you can trivially fit many models.
- Database supports advanced queries (e.g. find all results, where this parameter is in this range).
- Results use memory-light Python generators.

You can therefore fit (very) large datasets on a HPC and access the results efficiently via a Jupyter notebook.

```
agg = af.Aggregator.from_database("database.sqlite")

bulge = agg.lens.bulge
agg_query = agg.query(bulge == LightDeVaucouleurs)

for samples in agg_query.values("samples"):

    print("Maximum Log Likelihood Instance:")
    print(samples.max_log_likelihood_instance)
```

Advanced Modeling Tools

Search Grid Search: Massively parallel grid searches of non-linear searches.

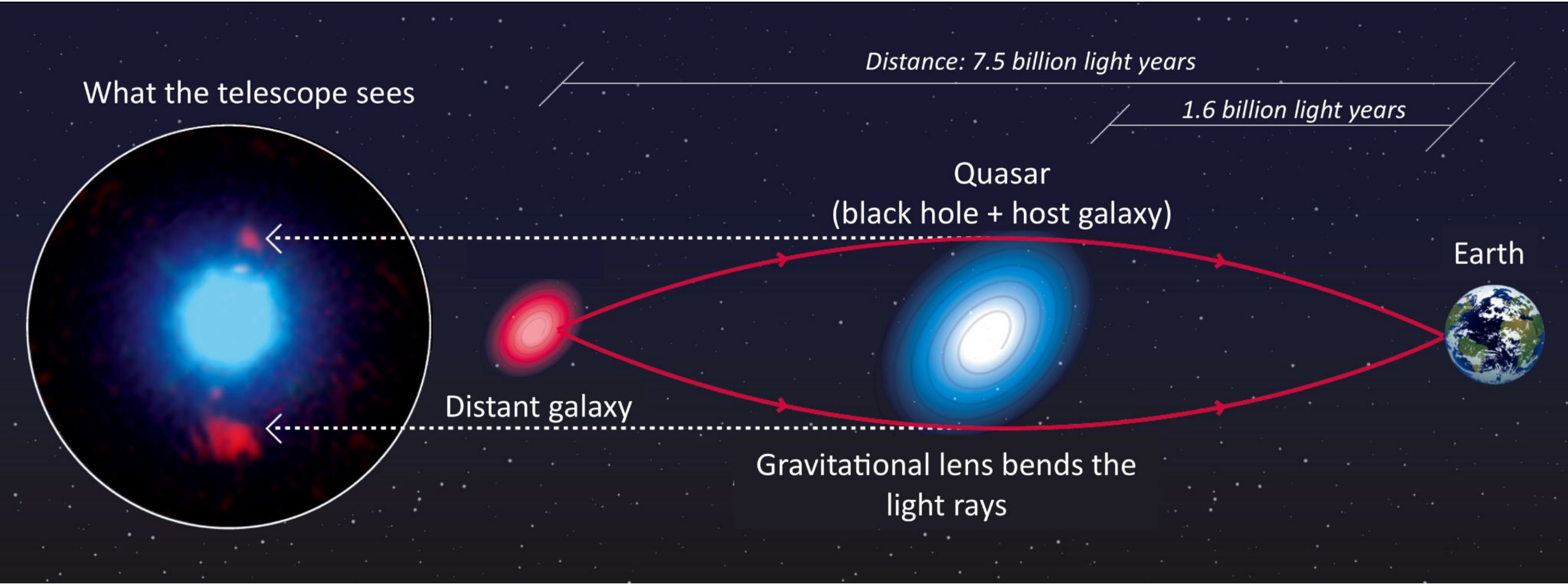
Search Chaining: Write highly customizable model-fitting pipelines that chain together multiple non-linear searches.

Sensitivity Mapping: Simulate and fit many datasets to determine when a more complex model would be accepted via model comparison.

Graphical / Hierarchical Models: Fit for global trends in large datasets by composing and fitting graphical models.

Cosmology: Strong Gravitational Lensing

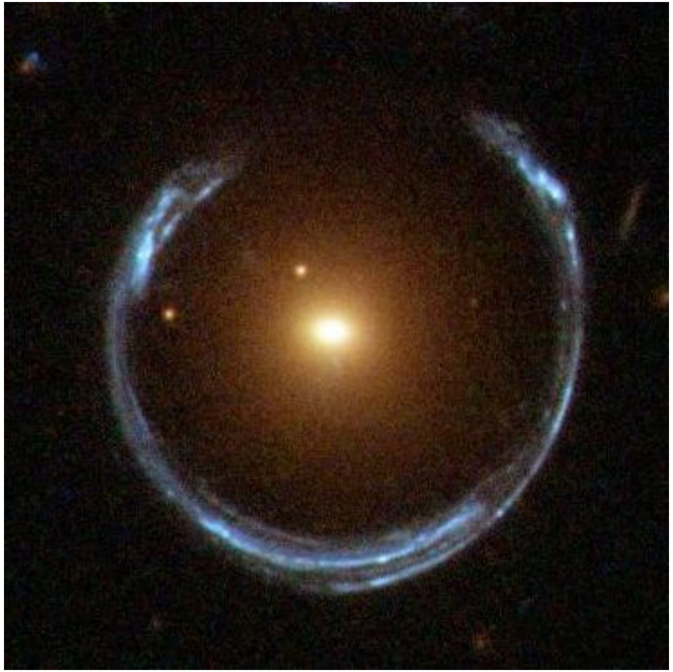
Strong Gravitational Lensing



Strong Gravitational Lensing

“Normal” Galaxy:

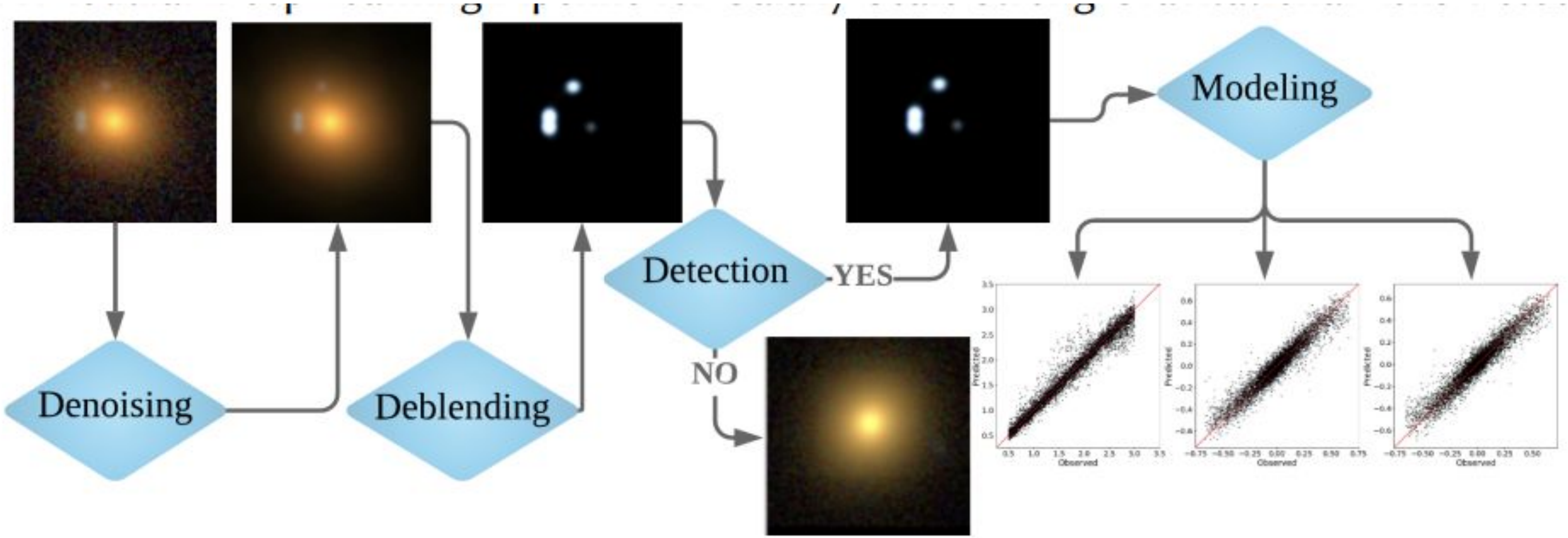
Strong Gravitational Lens:



Strong Gravitational Lensing Machine Learning

Growing literature on applying machine learning / CNN's to strong lens datasets.

- Can generate large training datasets cheaply.



PyAutoLens: Open Source Strong Gravitational Lensing

All code publically available (pip / conda), object oriented design, extensive documentation.

GitHub: https://github.com/Jammy2211/PyAutoLens
Readthedocs: https://pyautolens.readthedocs.io/en/latest/
JOSS paper: https://joss.theoj.org/papers/10.21105/joss.02825

The HowToLens Jupyter notebook lectures teach strong lens modeling to beginners (pitched at undergrads and above)!

HowToLens

Teach **anyone** how to model strong lenses with **PyAutoLens**.

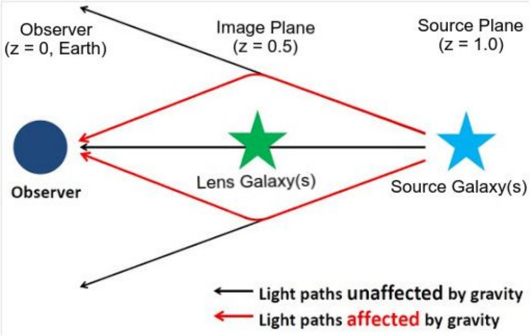
Perfect for **Level 4 students!**

Tutorial 4: Planes

So far, we have learnt how to combine light profiles, mass profiles and galaxies to perform various calculations. In this tutorial we'll use these objects to perform our first ray-tracing calculations!

A strong gravitational lens is a system where two (or more) galaxies align perfectly down our line of sight from Earth such that the foreground galaxy's mass (represented as mass profiles) deflects the light (represented as light profiles) of a background source galaxy(s).

When the alignment is just right and the lens is massive enough, the background source galaxy appears multiple times. The schematic below shows such a system, where light-rays from the source are deflected around the lens galaxy to the observer following multiple distinct paths.



As an observer, we don't see the source's true appearance (e.g. a round blob of light). Instead, we only observe its light after it has been deflected and lensed by the foreground galaxies.

In the schematic above, we used the terms 'image-plane' and 'source-plane'. In lensing, a 'plane' is a collection of galaxies at the same redshift (meaning that they are physically parallel to one another). In this tutorial, we'll use the `Plane` object to create a strong lensing system like the one pictured above. Whilst a plane can contain any number of galaxies, in this tutorial we'll stick to just one lens galaxy and one source galaxy.

In [ ]:
```
%matplotlib inline
from pyprojroot import here
workspace_path = str(here())
%cd $workspace_path
print(f"Working Directory has been set to `{workspace_path}`")

import autolens as al
import autolens.plot as aplt
```

Initial Setup

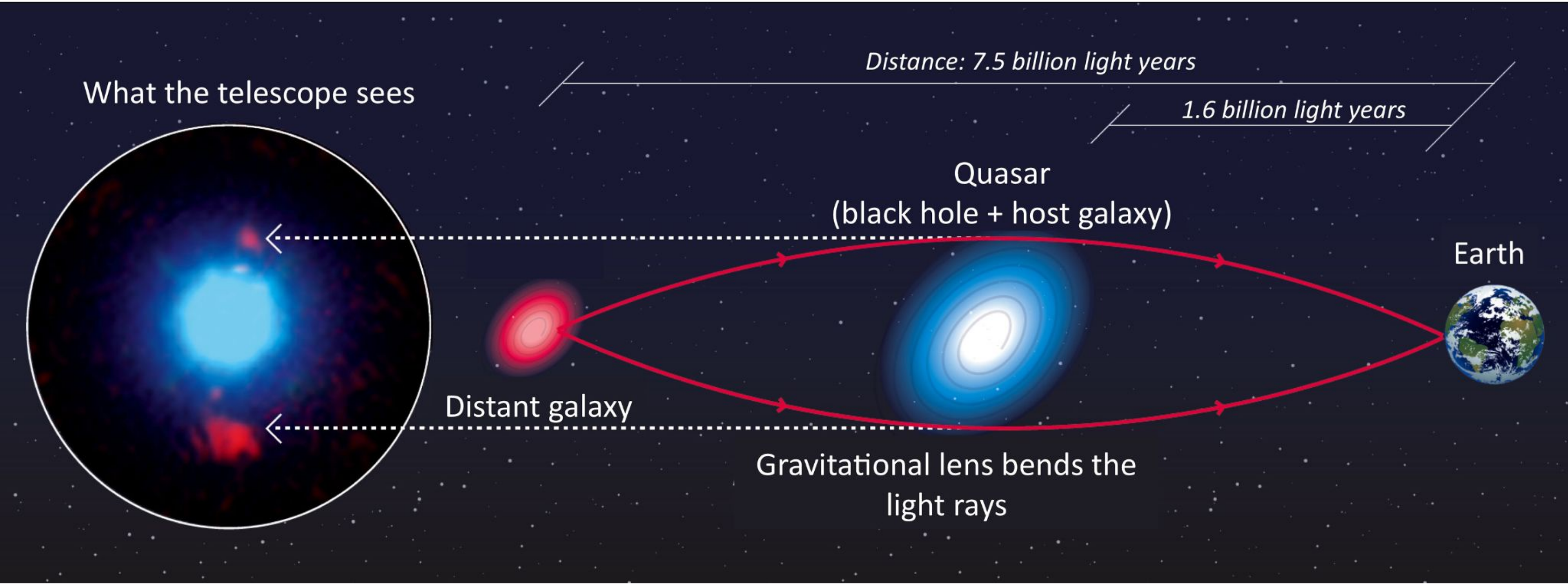
As always, we need a 2D grid of (y, x) coordinates.

However, we can now think of our grid as the coordinates that we are going to 'trace' from the image-plane to the source-plane. We name our grid the `image_plane_grid` to reflect this.

In [ ]:
```
image_plane_grid = al.Grid2D.uniform(shape_native=(100, 100), pixel_scales=0.05)
```

We will also name our `Galaxy` objects `lens_galaxy` and `source_galaxy`, to reflect their role in the schematic above.

Strong Gravitational Lensing



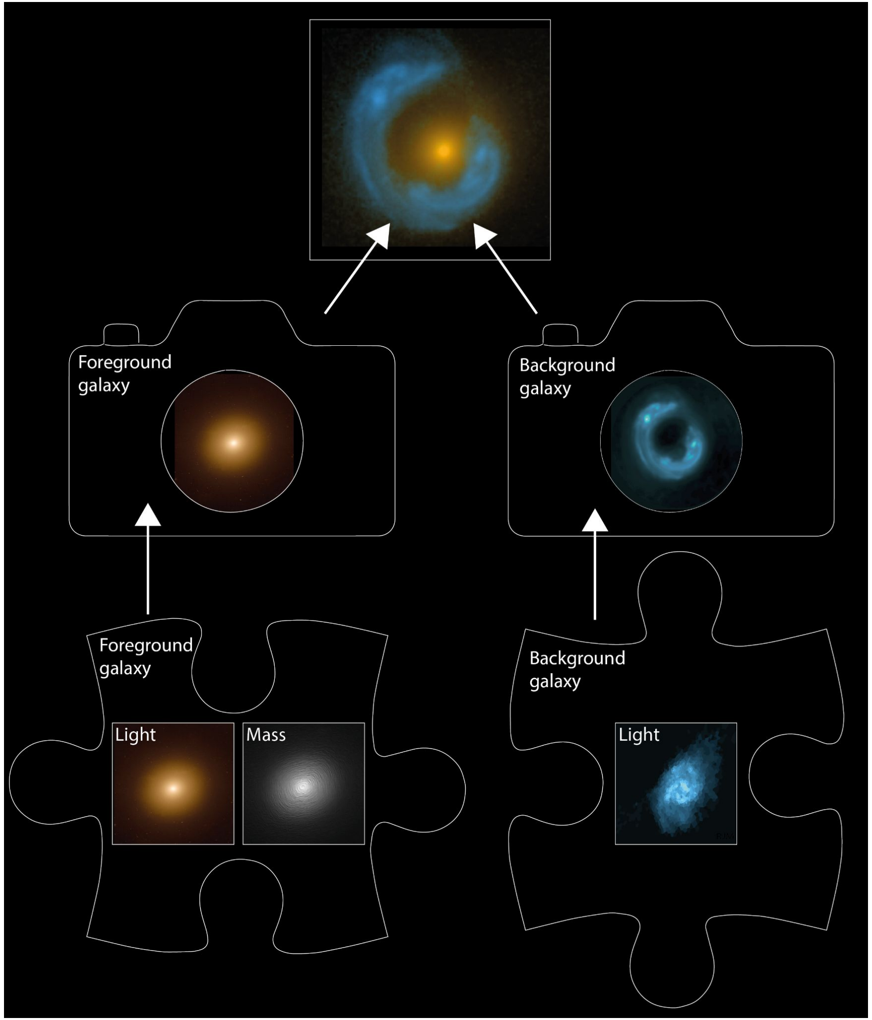
PyAutoFit: Model Composition

Model Composition

Break strong lens system into different **model components**:

Lens Galaxy: Light + Mass

Source Galaxy: Light



Light and Mass Profile classes

Write the **model components** of the problem as **Python classes** using the same API shown previously.

Note how the **model specific** calculations of this problem are functions of the classes.

Light Profile:

```
class LightDeVaucouleurs:

    def __init__(
        self,
        centre: typing.Tuple[float, float] = (0.0, 0.0),
        axis_ratio : float = 1.0,
        angle : float = 0.0,
        intensity: float = 0.1,
        effective_radius: float = 0.6,
    ):
        """The De Vaucouleurs light profile representing the bulge of galaxies...."""

        self.centre = centre
        self.axis_ratio = axis_ratio
        self.angle = angle
        self.intensity = intensity
        self.effective_radius = effective_radius

    def transform_grid_to_reference_frame(self, grid : np.ndarray):...

    def grid_to_elliptical_radii(self, grid : np.ndarray) -> np.ndarray:...

    def image_from_grid(self, grid : np.ndarray) -> np.ndarray:...
```

Light and Mass Profile classes

Mass Profile:

```python
class MassIsothermal:

    def __init__(
        self,
        centre: typing.Tuple[float, float] = (0.0, 0.0),
        axis_ratio : float = 1.0,
        angle : float = 0.0,
        mass: float = 1.0,
    ):
        """Represents an elliptical isothermal mass distribution...."""

        self.centre = centre
        self.axis_ratio = axis_ratio
        self.angle = angle
        self.mass = mass

    def transform_grid_to_reference_frame(self, grid : np.ndarray):...

    def rotate_grid_from_reference_frame(self, grid : np.ndarray) -> np.ndarray:...

    def psi_from(self, grid : np.ndarray) -> np.ndarray:...

    def deflections_from_grid(self, grid : np.ndarray) -> np.ndarray:...
```

Light Profile:

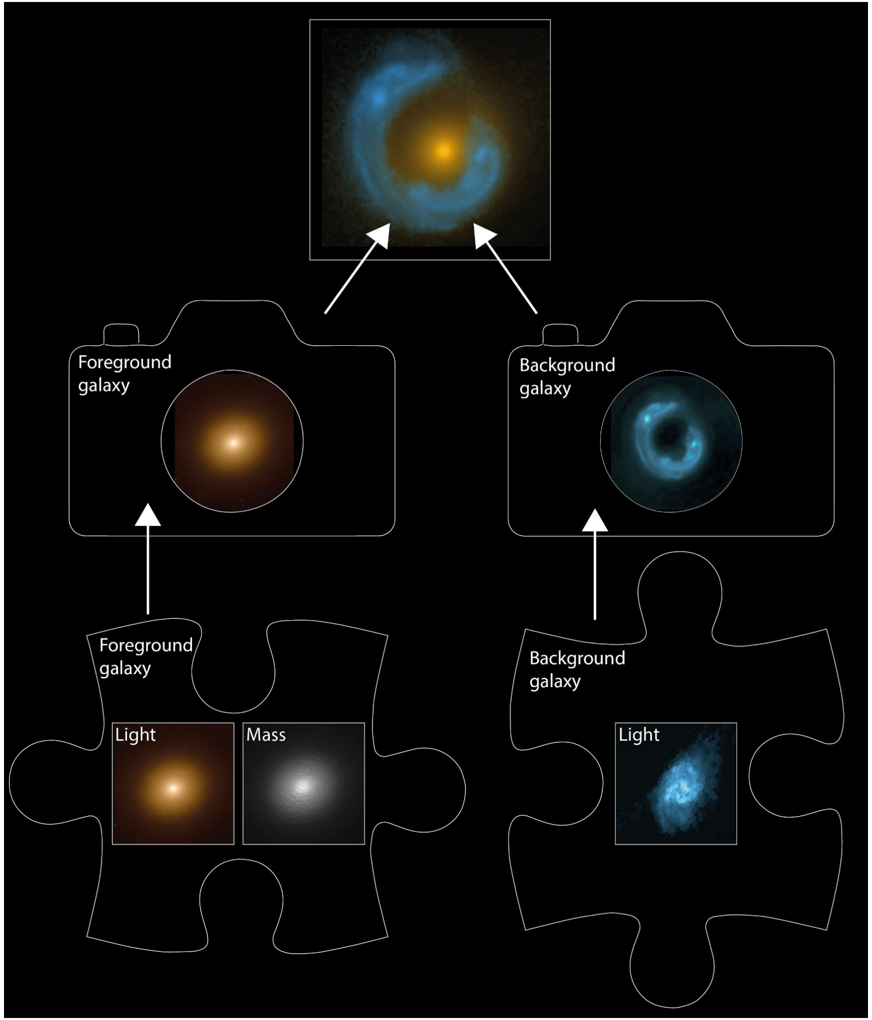
```python
class LightExponential:

    def __init__(
        self,
        centre: typing.Tuple[float, float] = (0.0, 0.0),
        axis_ratio : float = 1.0,
        angle : float = 0.0,
        intensity: float = 0.1,
        effective_radius: float = 0.6,
    ):
        """The Exponential light profile representing the disk of galaxies...."""

        self.centre = centre
        self.axis_ratio = axis_ratio
        self.angle = angle
        self.intensity = intensity
        self.effective_radius = effective_radius

    def transform_grid_to_reference_frame(self, grid : np.ndarray) -> np.ndarray:...

    def grid_to_elliptical_radii(self, grid : np.ndarray) -> np.ndarray:...

    def image_from_grid(self, grid : np.ndarray) -> np.ndarray:...
```

Model Composition

Break strong lens system into different **model components**:

Lens Galaxy: Light + Mass

Source Galaxy: Light



Python Classes

The use of Python Classes to define the has a crucial additional benefit.

- It allows for **multi-level model composition.**

Core for PyAutoFit's graphical modeling and hierarchical modeling functionality.

Galaxy Class

Combine the mass and light profiles **at a specific redshift** to make the lens galaxy and source galaxy.

Note how the **image_from_grid** and **deflections_from_grid** methods are included, which use the methods of the individual light and mass profiles.

Redshift = Distance from us in the Universe.

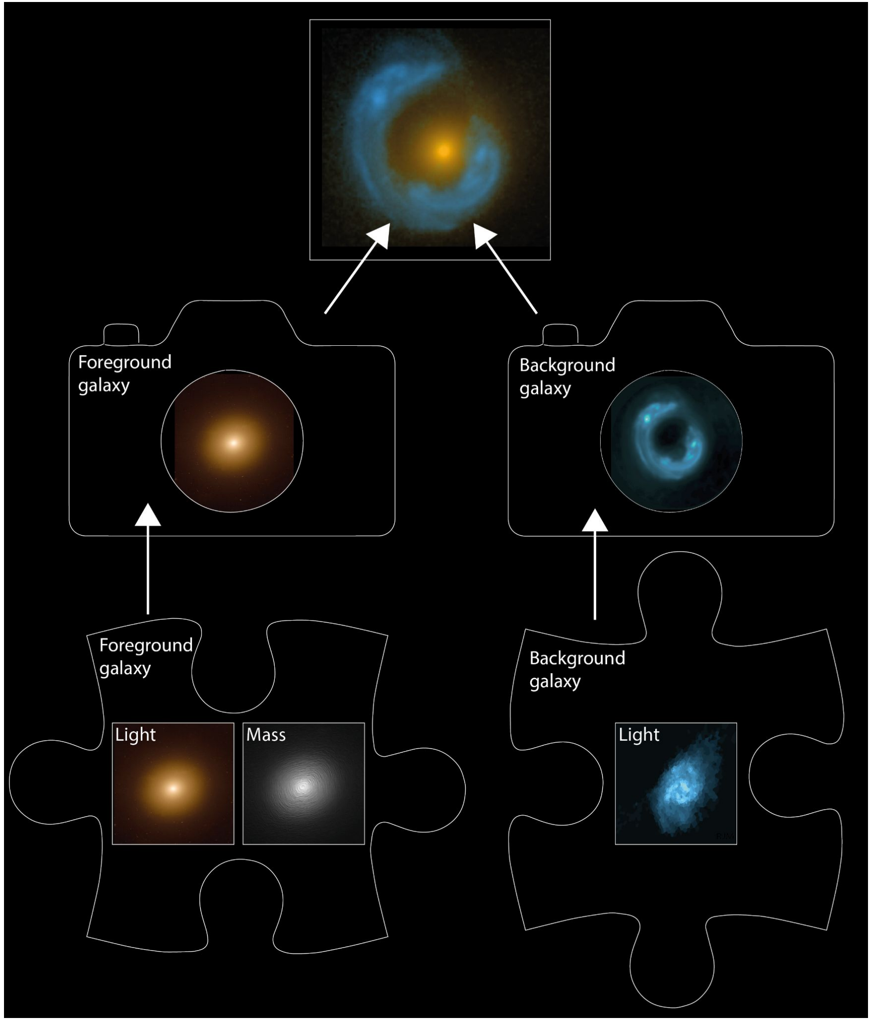
```python
class Galaxy:

    def __init__(
            self,
            redshift: float,
            light_profiles: Optional[List] = None,
            mass_profiles: Optional[List] = None,
    ):
        """A galaxy, which contains light and mass profiles at a specified redshift...

        self.redshift = redshift
        self.light_profiles = light_profiles
        self.mass_profiles = mass_profiles

    def image_from_grid(self, grid : np.ndarray) -> np.ndarray:
        """..."""
        if len(self.light_profiles) > 0:
            return sum(
                map(lambda p: p.image_from_grid(grid=grid), self.light_profiles)
            )
        return np.zeros((grid.shape[0],))

    def deflections_from_grid(self, grid : np.ndarray) -> np.ndarray:
        """..."""
        if len(self.mass_profiles) > 0:
            return sum(
                map(lambda p: p.deflections_from_grid(grid=grid), self.mass_profiles)
            )
        return np.zeros((grid.shape[0], 2))
```

Model Composition

Break strong lens system into different **model components**:

Lens Galaxy: Light + Mass

Source Galaxy: Light



Composing the Model

Scans every light and mass profile to determine this model has 16 free parameters that the non-linear search fits.

- A user can easily extend the model with more light profiles, mass profiles, etc.

This is the API a user of your model-fitting software is greeted with!

```
import autofit as af

lens_galaxy_model = af.Model(
    Galaxy,
    redshift=0.5,
    bulge=LightDeVaucouleurs,
    mass=MassIsothermal
)
source_galaxy_model = af.Model(
    Galaxy,
    redshift=1.0,
    disk=LightExponential
)

model = af.Collection(
    lens=lens_galaxy_model,
    source=source_galaxy_model
)
```

Writing the Analysis

By using Python classes as the model components, this means we can write a concise likelihood function.

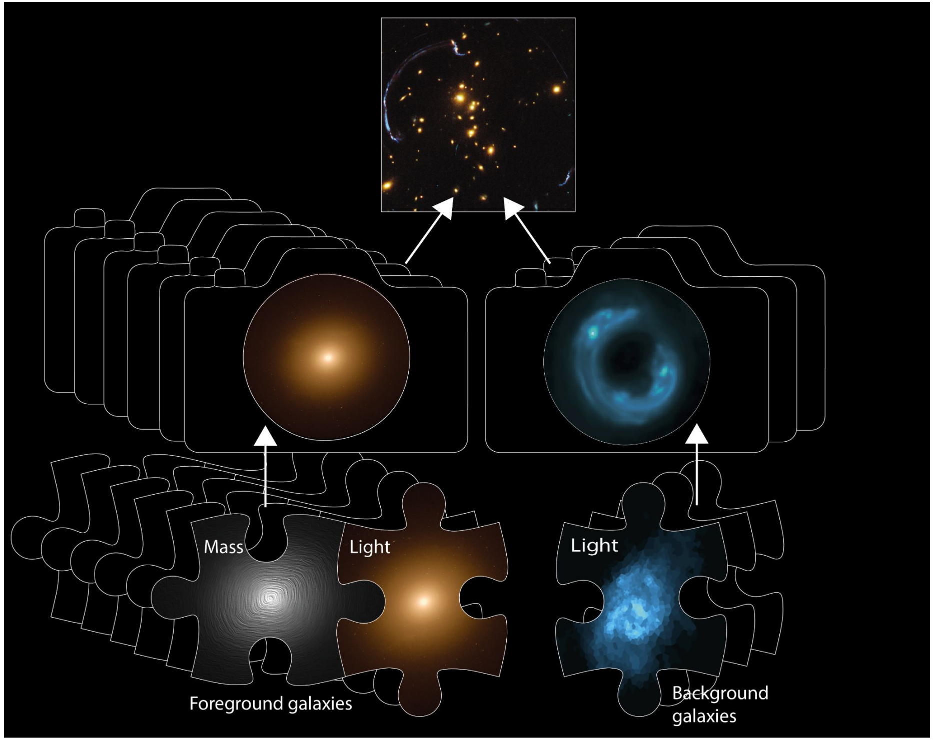
- Cleanly separate the model-specific code (e.g. light profiles, mass profiles, lensing) from the model-fitting code.
- Easy to extend and customize the Analysis class for bespoke model-fitting.

```python
class Analysis(af.Analysis):

    def __init__(self, image, noise_map, psf, grid):

        self.image = image
        self.noise_map = noise_map
        self.psf = psf
        self.grid = grid

    def log_likelihood_function(self, instance):

        """
        The 'instance' that comes into this method contains the `Galaxy`'s
        we setup in the model.
        """

        print("Lens Model Instance:")
        print("Lens Galaxy = ", instance.lens)
        print("Lens Galaxy Bulge = ", instance.lens.bulge)
        print("Lens Galaxy Bulge Centre = ", instance.lens.bulge.centre)
        print("Lens Galaxy Mass Centre = ", instance.lens.mass.centre)
        print("Source Galaxy = ", instance.sources)

        """
        The methods of the `Galaxy` class are available, making it easy to fit
        the lens model.
        """

        lens_image = instance.lens.image_from_grid(grid=self.grid)
        deflections = instance.lens.deflections_from_grid(grid=self.grid)
        source_grid = self.grid - deflections
        source_image = instance.source.image_from_grid(grid=source_grid)

        model_image = lens_image + source_image
        model_image = self.psf.convolve(model_image)

        residual_map = self.image - model_image
        chi_squared_map = (residual_map / self.noise_map) ** 2.0
        log_likelihood = -0.5 * sum(chi_squared_map)

        return log_likelihood
```

Model Composition

Straightforward for complex models to be composed and fitted in a scalable and streamlined way:

- Easy to extend model galaxies with many light and mass profiles.
- Or extend the model with many more galaxies.

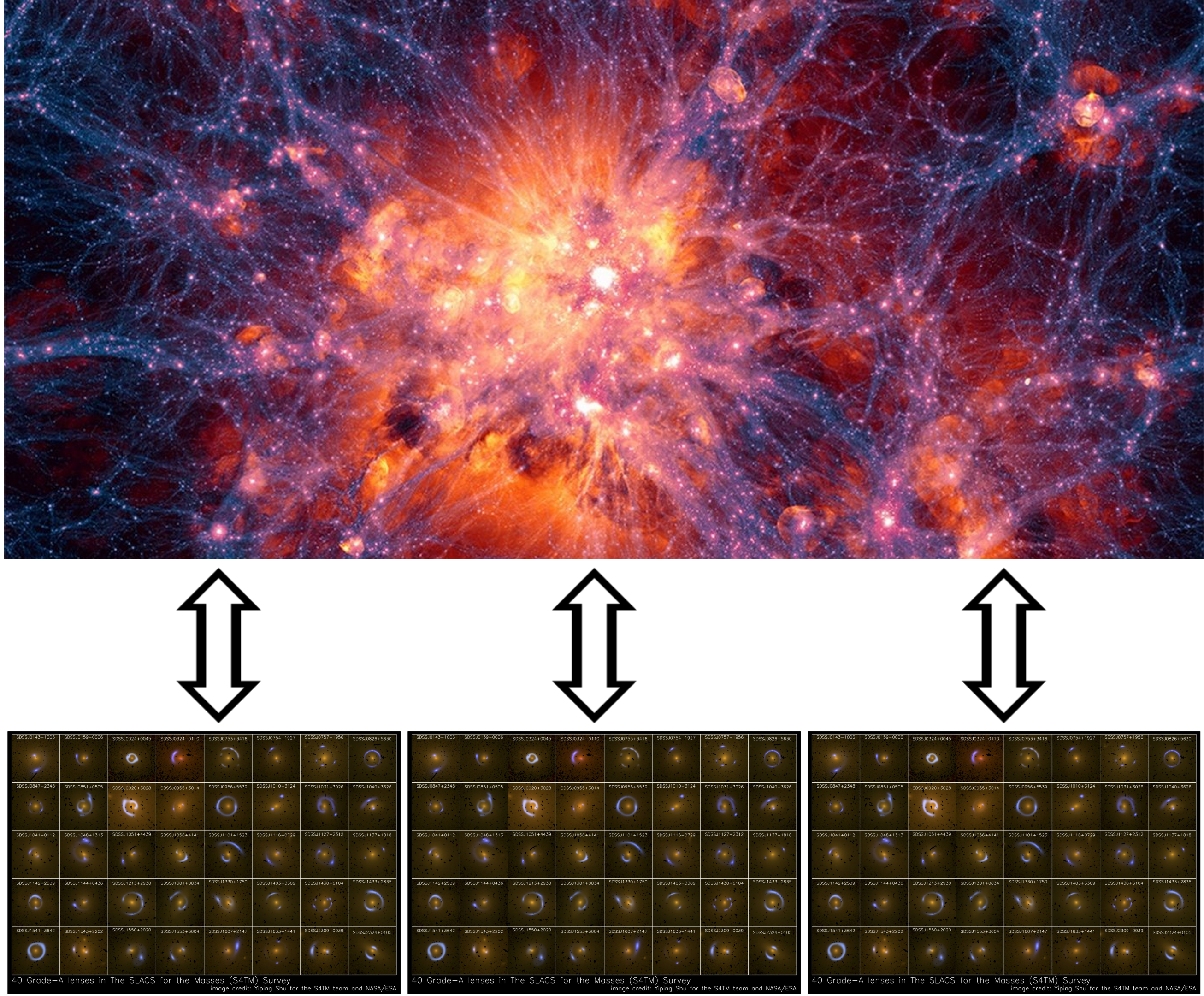


PyAutoFit: Graphical Models

Hierarchical Statistical Models

Perform **detailed** modeling of every individual galaxy.

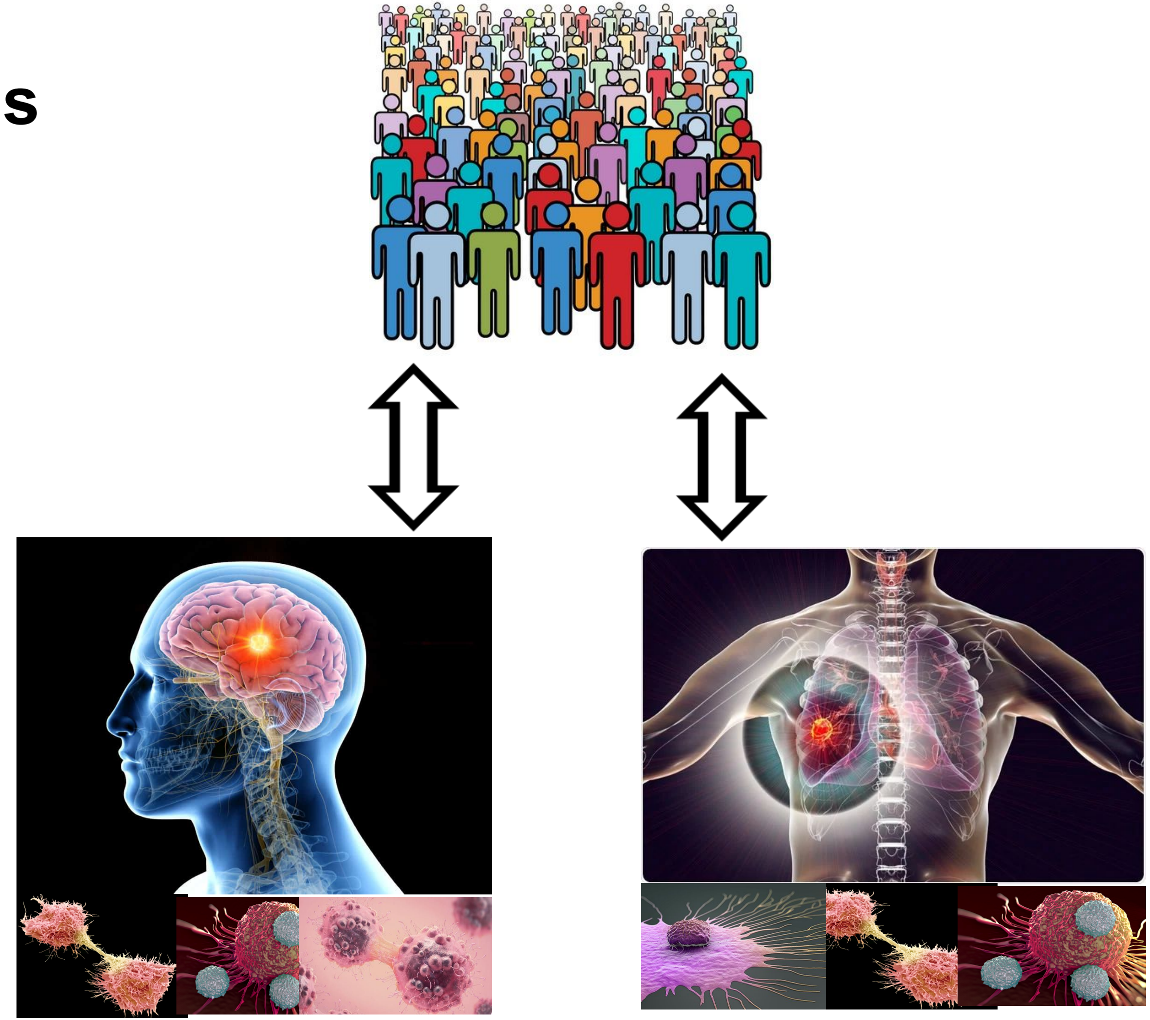
For **large datasets flow information up** to learn about the Universe.



Cancer in Populations

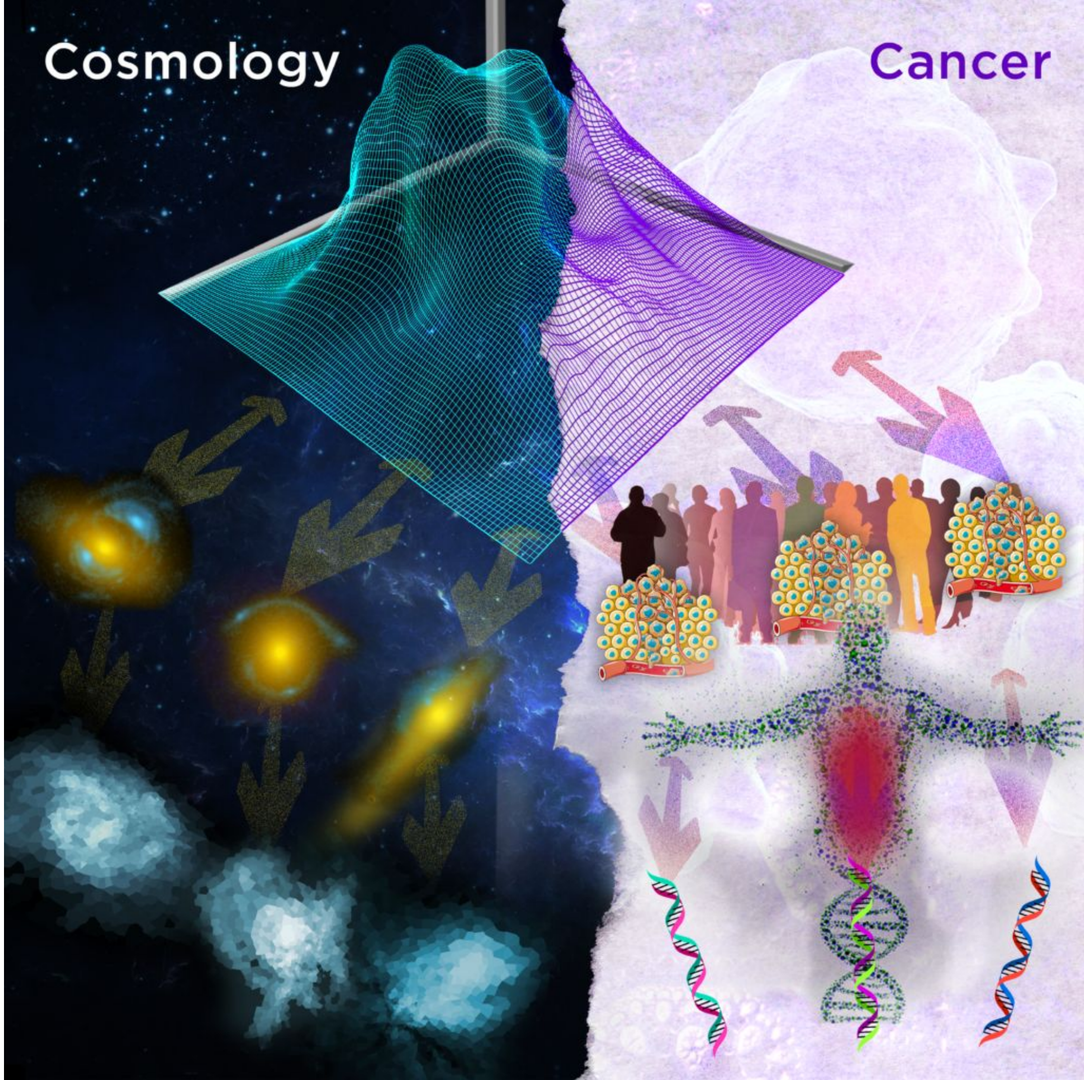
We build a **detailed** model **of every individual cancer.**

Each one tells us **extremely small amount of information about the dynamics of that cancer across a population.**



Cosmology & Cancer

Performing **detailed analysis** of **large** datasets, whilst extracting a **small** amount of information from each about a **global model**.



Summary

The statistical techniques used to understand the Universe's **Cosmology** can be applied to studies of **Cancer**.

PyAutoFit is an open source project with scope way beyond just studies by Astronomers and healthcare scientists!

GitHub: https://github.com/Jammy2211/PyAutoLens
Readthedocs: https://pyautolens.readthedocs.io/en/latest/
JOSS paper: https://joss.theoj.org/papers/10.21105/joss.02550
Binder: https://mybinder.org/v2/gh/Jammy2211/autofit_workspace/HEAD

PyAutoFit: Cosmology & Cancer

ConcR

Deep Science Ventures Biotech company

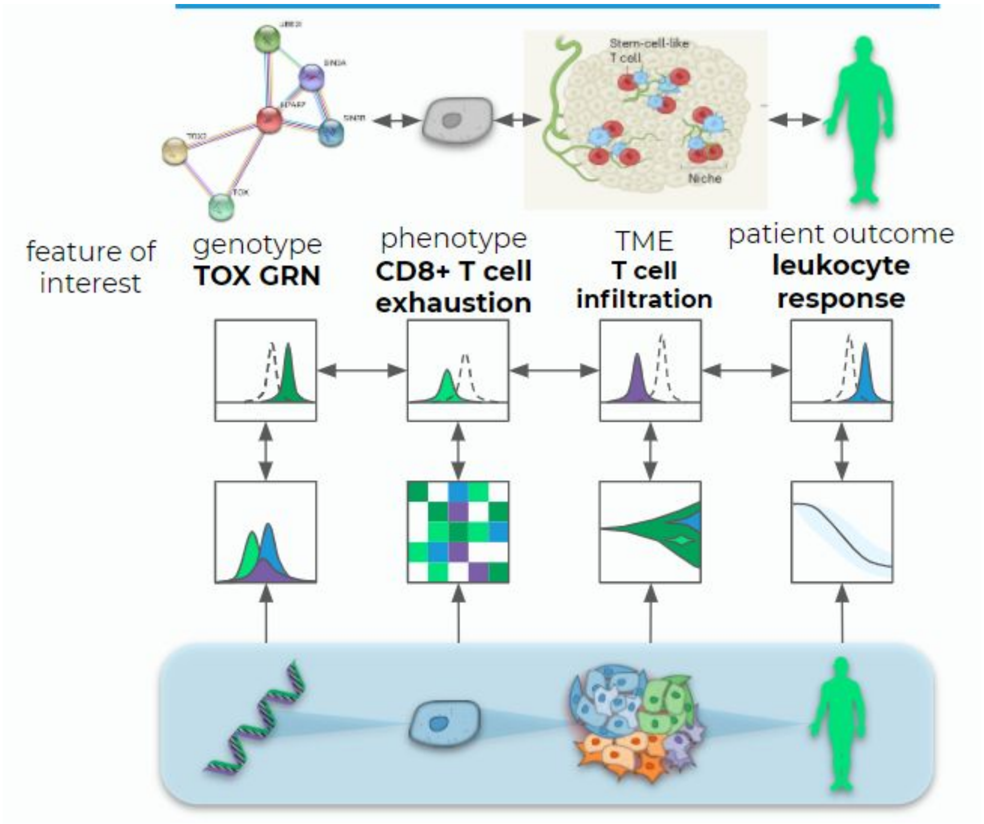
https://www.concr.co/

STFC Opportunities Grant and Innovate UK grant with Roche, UCL and others.

Cancer Cell Model

Multi-level model of cancer growth:

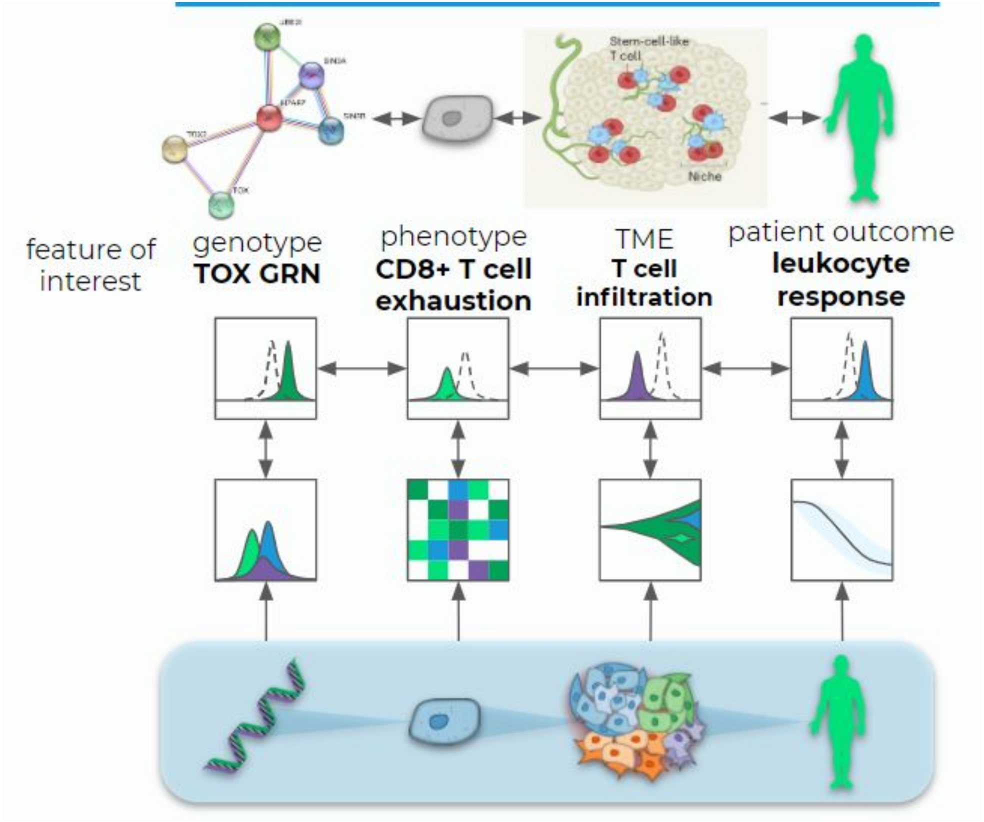
- Separate multiscale complexity of cancer model describing specific sub-behaviours.
- For example, a specific model of how a cell with a specific genetic or epigenetic profiles responds to treatment.



Cancer Tumour Model

Multi-level model of cancer growth:

- The dynamics of a tumour are then modeled as subpopulations of cells with their own genetic and cellular profiles.
- This acts as a **high level model** that includes the interactions between these different cells.



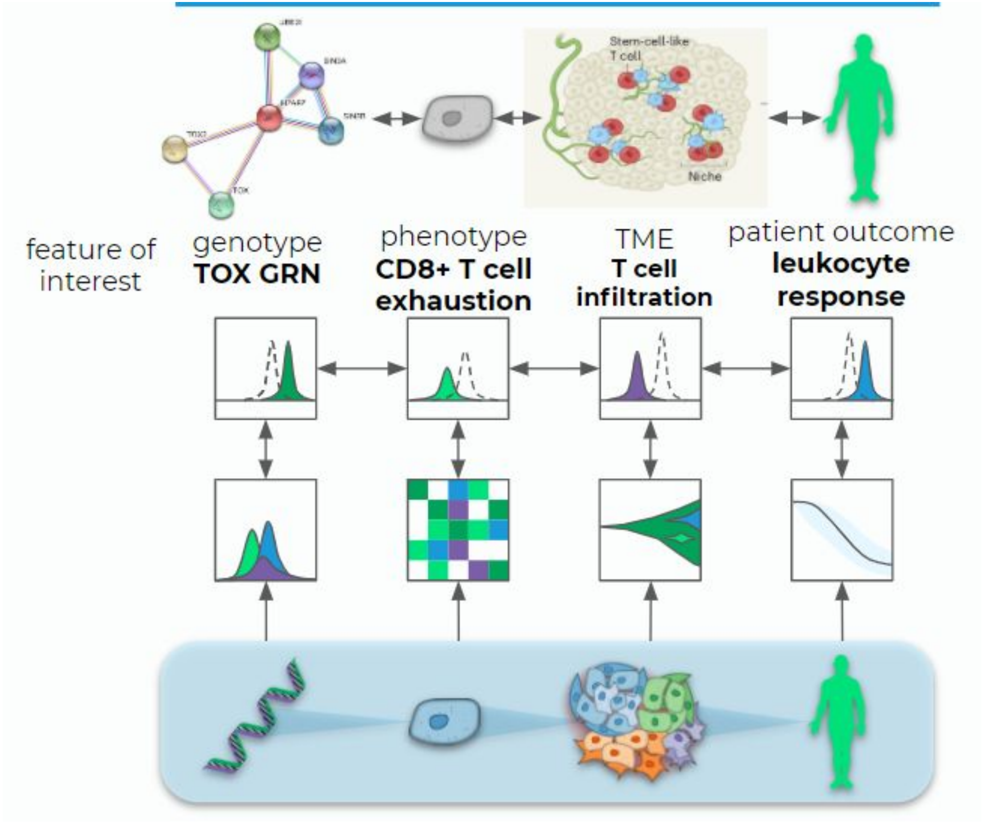
Patient Outcome

Multi-level model of cancer growth:

- The health of the patient is then governed by the dynamics of multiple tumours and their interactions with the patient.
- This is another **higher level model**.

The major goal is to perform this analysis on samples of **many patients** creating a high dimensionality multi-level model.

A variety of **datasets** are used for this analysis, will detail in a moment.



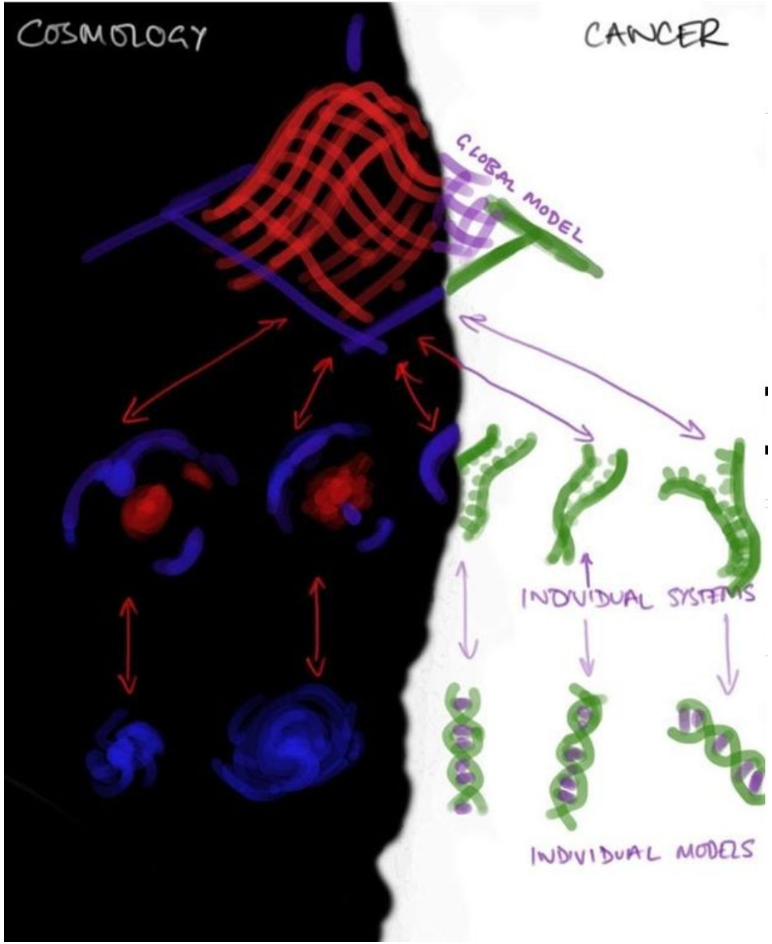
Cosmology & Cancer

We began talking to ConcR about 1 year into PyAutoFit develop, saw obvious overlap:

- Composition of multi-level models in a scalable way.
- Customization in how the model is fitted.
- Customization in how different datasets are fitted.
- Same challenge in terms of big data, massively parallel computing.

However, a core feature was missing at the heart of ConcR's modeling:

- Ability to fit multi-level models in a way that can scale up to 100000+ parameters.
- API for multi-level modeling was non-existent.


COSMOLOGY
CANCER
GLOBAL MODEL
INDIVIDUAL SYSTEMS
INDIVIDUAL MODELS

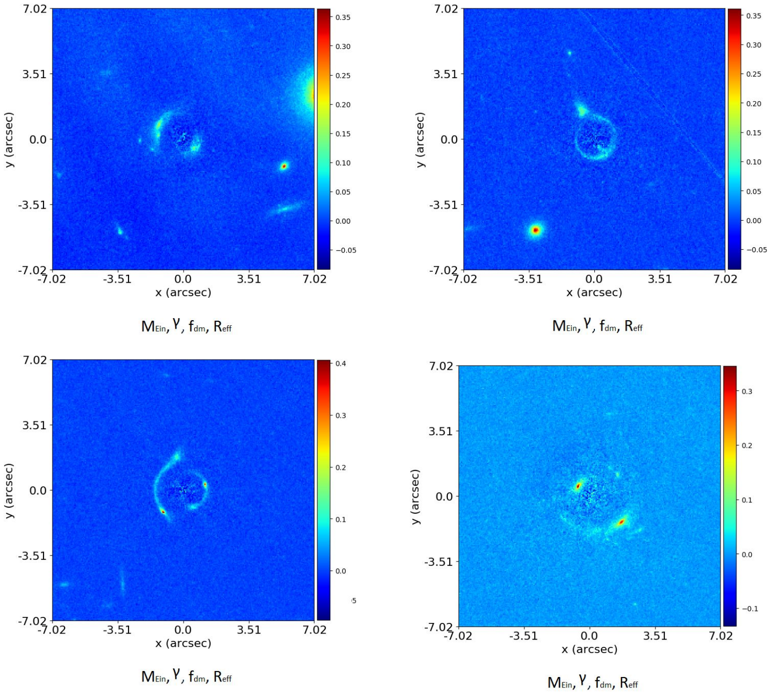
Strong Lens Modeling

The focus of my research was fitting **individual strong lenses.**

- Turns this is a really hard problem, but one we have now solved.

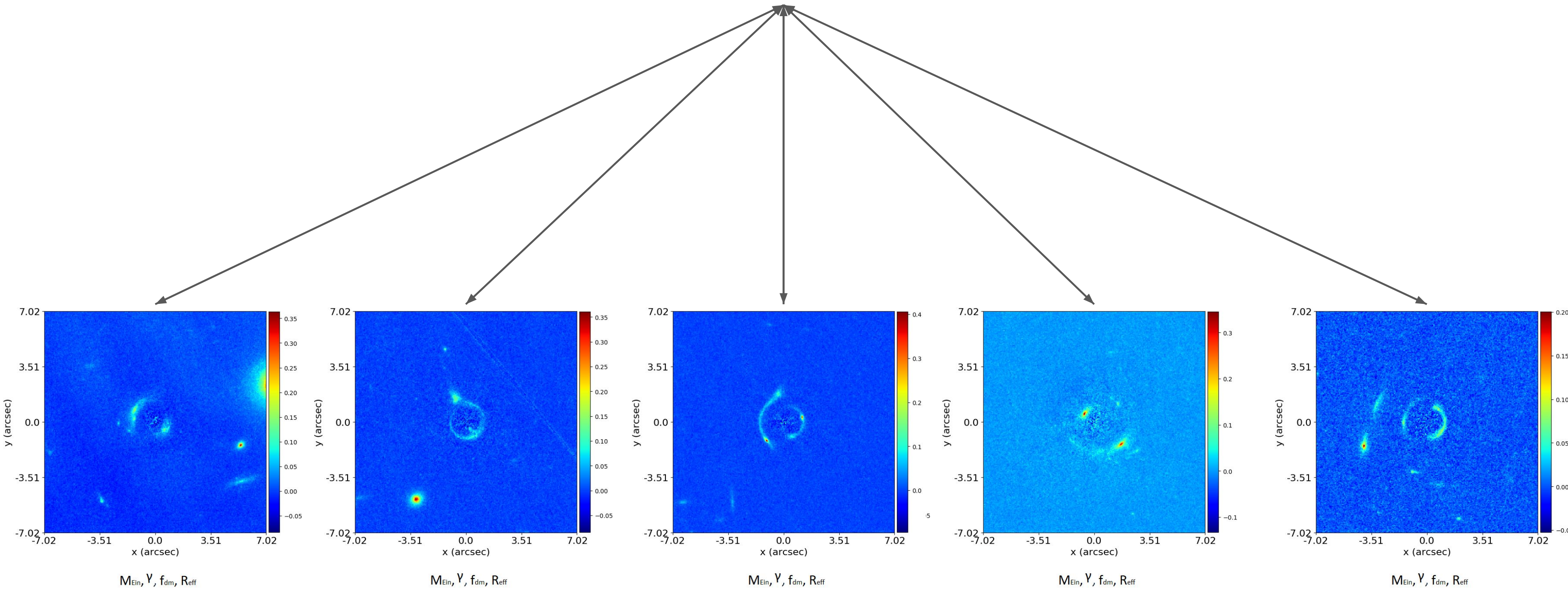
Always knew we would need to fit *thousands of lenses* learn about **galaxy formation and cosmology.**

This requires a **multi-level model.**



Graphical and Hierarchical Models

$$f\,(M_{Ein}, \gamma', f_{dm}, R_{eff}, \ldots)$$



Expectation Propagation

Even with an API for composing multi-level models, how do we actually fit them?

- Each dataset has 5+ data specific parameters.
- For the problems we're talking about, this could lead to models with 500000+ parameters.
- Cannot fit using traditional non-linear search due to **curse of dimensionality.**

Expectation Propagation

Even with an API for composing multi-level models, how do we actually fit them?

- Each dataset has 5+ data specific parameters.
- For the problems we're talking about, this could lead to models with 500000+ parameters.
- Cannot fit using traditional non-linear search due to **curse of dimensionality.**

Building expectation propagation based model-fitting into graphical models (https://arxiv.org/abs/1412.4869**).**

- Fits individual nodes of the graphical model, one-by-one, in parameter spaces of reduced dimensionality.
- Pass information 'up' and 'down' scouring the model graph, working our way up to the high-level model components.

Example Graphical Model

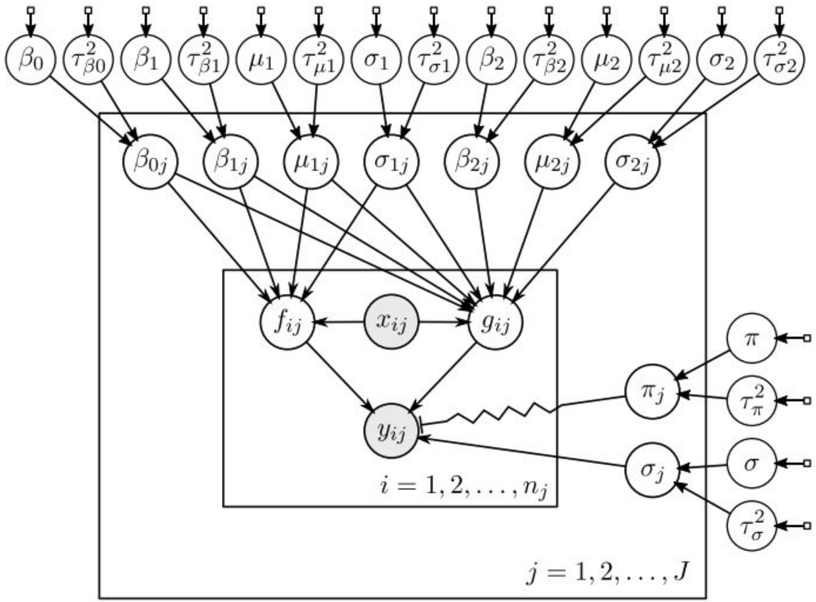


Figure 10: Graphical representation of the astronomy model. Circles represent random variables and boxes represent fixed parameters. Grayed circles are observed. The zig-zag line indicates that π_j functions as a selector between f_{ij} and g_{ij}. The labels for the fixed prior parameters are omitted for clarity.

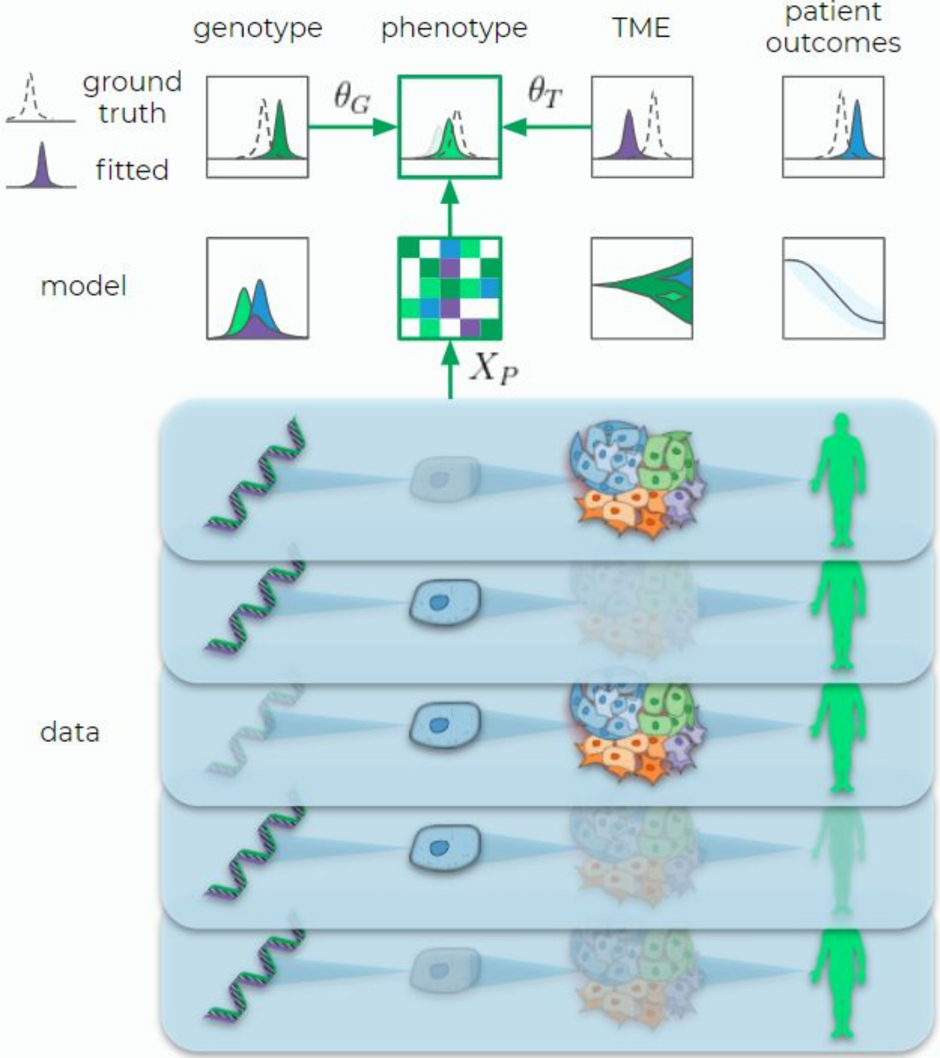
Cancer Datasets

Many different types of data are fitted:

- genomic sequencing, cellular imaging, radioimaging, health outcomes, clinical trial data, etc.

On the ‘good side’ of quality / homogeneity for health datasets (no reliance on doctor’s scribbles).

A core selling point of ConcR, via the graphical modeling, is that it can account for missing data.



PyAutoFit: A PPL for Astronomers

The interface of other PPLs is designed for topological models, integrating functions, thermodynamic parameter spaces, etc.

- Great for statisticians, theorists and those with a **certain types** of model-fitting problems.

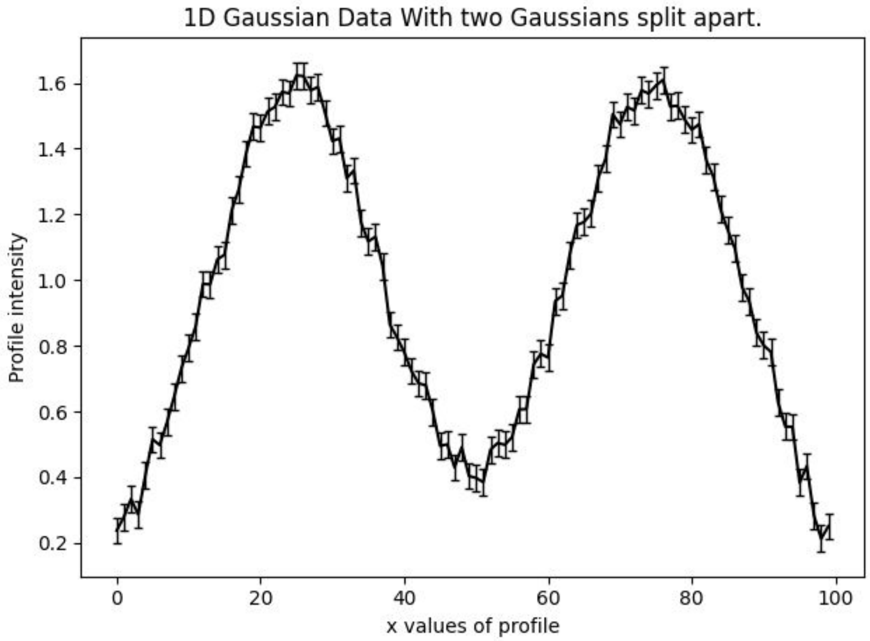
- Not so great for those with more **data focused** model-fitting problems**.**

```python
import pymc3 as pm

X, y = linear_training_data()
with pm.Model() as linear_model:
    weights = pm.Normal("weights", mu=0, sigma=1)
    noise = pm.Gamma("noise", alpha=2, beta=1)
    y_observed = pm.Normal(
        "y_observed",
        mu=X @ weights,
        sigma=noise,
        observed=y,
    )

    prior = pm.sample_prior_predictive()
    posterior = pm.sample()
    posterior_pred = pm.sample_posterior_predictive(posterior)
```

PyAutoFit: Advanced Features

Search Chaining

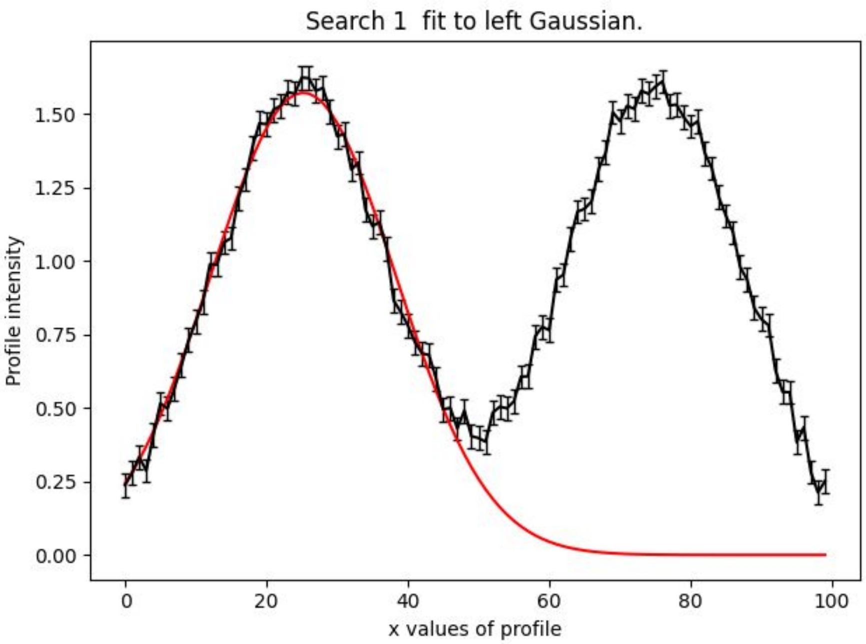
Break a model-fit into a **chained sequence** of searches:



Search Chaining

Break a model-fit into a chained sequence of searches:

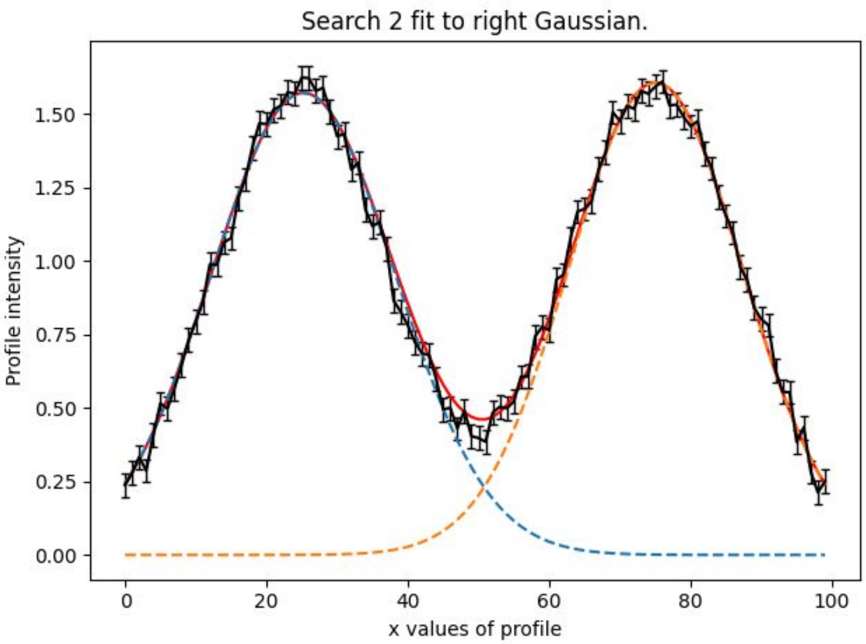
- Search 1: fit model to left Gaussian with **fast non-linear search**.



Search Chaining

Break a model-fit into a chained sequence of searches:

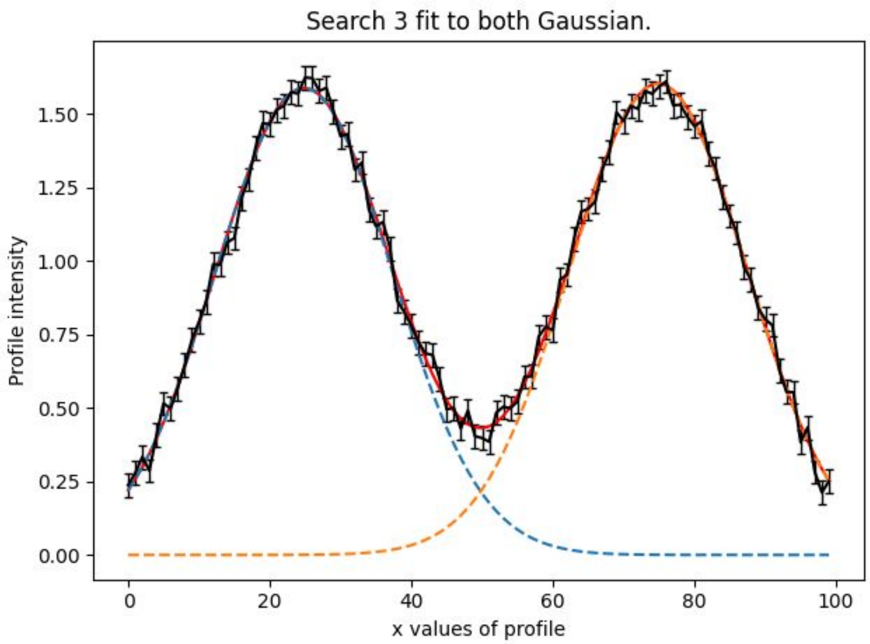
- Search 1: fit model to left Gaussian with **fast non-linear search**.
- Search 2: fit model to right Gaussian with fast non-linear search and result of search 1.



Search Chaining

Break a model-fit into a chained sequence of searches:

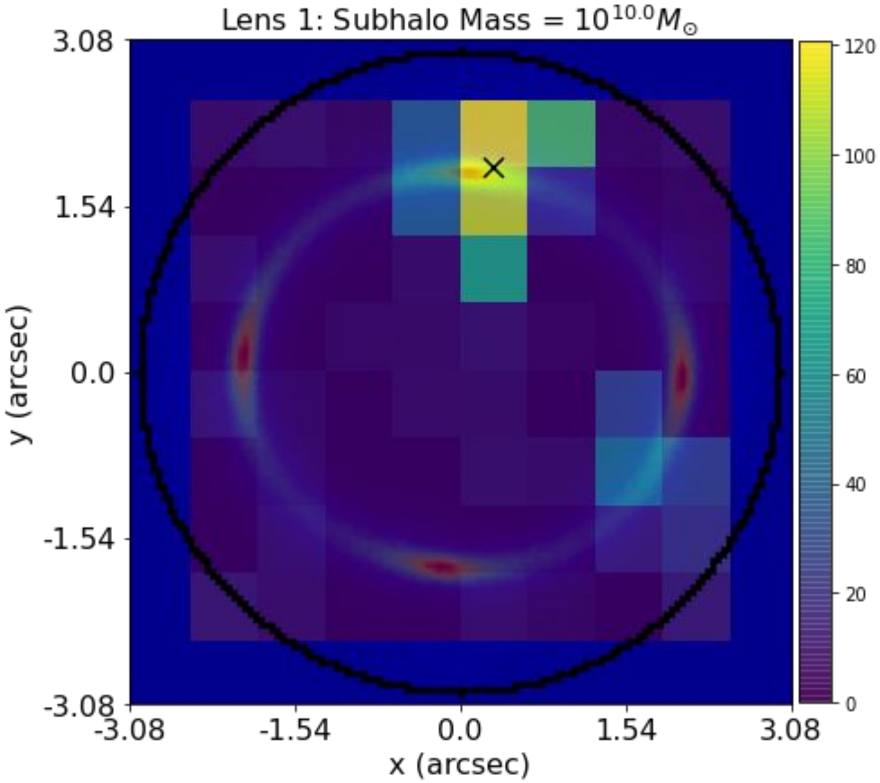
- Search 1: fit model to left Gaussian with **fast non-linear search**.
- Search 2: fit model to right Gaussian with **fast non-linear search** and result of search 1.
- Search 3: Fit both Gaussians simultaneously with **thorough non-linear search and a parameter space starting point** inferred from first two searches.



Grid Search of Non-linear Searches

Break a model-fit into a grid search of searches:

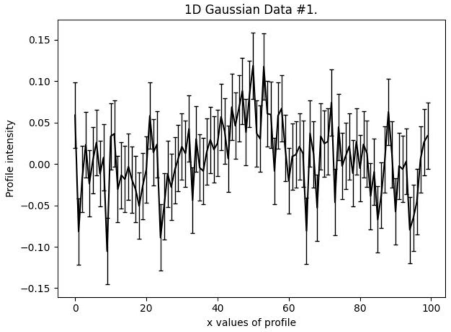
- Support for massively parallel fits.
- Database provides tools for analysing results efficiently.

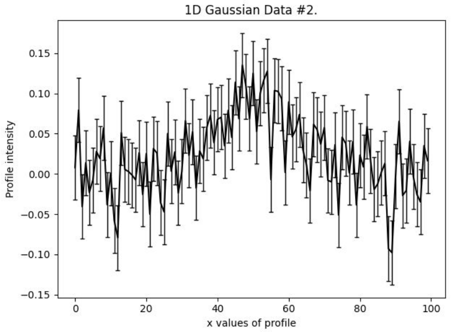


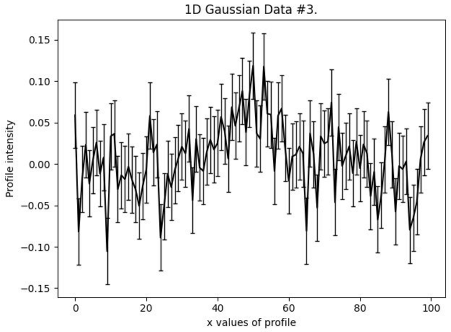
Graphical and Hierarchical Models

Compose multi-level models for fitting many datasets:

- Simple example of fitting three low signal to noise Gaussians simultaneously.
- Can assume all three Gaussians have same centre, but different intensity / sigma.







Graphical and Hierarchical Models

Can set up unique **Analysis** class, which will pair every dataset with components of the multi-level model:

```
analysis_0 = a.Analysis(data=data_0, noise_map=noise_map_0)
analysis_1 = a.Analysis(data=data_1, noise_map=noise_map_1)
analysis_2 = a.Analysis(data=data_2, noise_map=noise_map_2)
```

Model is built out of individual model components like before.

```
centre_shared_prior = af.GaussianPrior(mean=50.0, sigma=30.0)

gaussian_0 = af.Model(m.Gaussian)
gaussian_0.centre = centre_shared_prior
gaussian_0.intensity = af.GaussianPrior(mean=10.0, sigma=10.0)
gaussian_0.sigma = af.GaussianPrior(mean=10.0, sigma=10.0)  # This prior is used by all 3 Gaussians!

prior_model_0 = af.Collection(gaussian=gaussian_0)

gaussian_1 = af.Model(m.Gaussian)
gaussian_1.centre = centre_shared_prior
gaussian_1.intensity = af.GaussianPrior(mean=10.0, sigma=10.0)
gaussian_1.sigma = af.GaussianPrior(mean=10.0, sigma=10.0)  # This prior is used by all 3 Gaussians!

prior_model_1 = af.Collection(gaussian=gaussian_1)
```