Generic bias in BORG

- People will want to try many bias models
- I have tried to factorize the concept of bias in the likelihood framework

The conceptual framework

Selection function modifier

$$\mathcal{L}(N^{\rm obs}|\rho_m) = \mathcal{L}_j^{\rm elem}(\{N_j^{\rm obs}\}, \bar{S}(\{b_i(\rho_m)\}_i))$$

The total final likelihood

The elementary likelihood (i.e. pure Gauss)

Biased fields (including noise...)

Example on linear Gaussian

The elementary likelihood

$$\mathcal{L}^{\mathrm{elem}}(N^{\mathrm{obs}}, N^{\mathrm{pred}}, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(N^{\mathrm{obs}} - N^{\mathrm{pred}})^2}{\sigma^2}\right)$$

The selection function dressing

$$S(N^g, \sigma) = \begin{pmatrix} S_0(N^g, \sigma) \\ S_1(N^g, \sigma) \end{pmatrix}$$

$$S_{0,i}(N^g, \sigma) = R_i N_i^g$$

$$S_{1,i}(N^g, \sigma) = R_i \sigma_i^2$$

Linear bias model

$$N_i^g = A\rho_{m,i}^{\alpha}$$

$$\sigma_i = \sigma$$

The elementary likelihood model

This is implemented as a “Protocol” (gaussian.hpp):

```
class Gaussian {

  static const size_t numberLikelihoodParams = 2;

  static auto sample(RandomGen& rgen, TupleLike tuple_data);

  static double log_probability(const DataArray& data, TupleLike tuple_data,
MaskArray&& mask);

  static auto diff_log_probability(const DataArray& data, TupleLike tuple_data,
const Mask& mask);

};
```

Unfortunate remaining cooking

```
static double log_probability(const DataArray& data, TupleLike tuple_data, MaskArray&& mask)  {
   auto const& intensity = std::get<0>(tuple_data);
   auto const& r = std::get<1>(tuple_data);
   double Nmask = 0;

    // Do a sum over voxels not masked.
    Nmask = LibLSS::reduce_sum<double>(
          b_va_fused<double>(
            [](double sigma2)->double {
              return std::log(sigma2);
            },
            r
        ),
        mask);

    double chi2 = -0.5*LibLSS::reduce_sum<double>(
          b_va_fused<double>(log_proba, data, intensity, r),
          mask
    );
    double N2 = -0.5*Nmask;
   return chi2 + N2;
}
```

This is the predicted signal array

This is the variance array

Parallel reduction over not masked pixels

Compute log of each element

Implement (x-y)/sigma^2

The power law bias function

```
struct PowerLaw {
 const auto numParams = 1;

  double alpha;
  double nmean;
  SimpleAdaptor selection_adaptor;

  void setup_default(BiasParameters& params) {
    params[0] = 1.5;
  }

  void prepare(ForwardModel& fwd_model, const FinalDensityArray& final_density,
    double const _nmean, const BiasParameters& params, MetaSelect _select = MetaSelect())

  void cleanup() {}

  static bool check_bias_constraints(Array&& a) {
    return a[0] > 0 && a[0] < 5; // alpha can take any value
  }

  auto compute_density(const FinalDensityArray& array);

  auto apply_adjoint_gradient(const FinalDensityArray& array,
    TupleGradientLikelihoodArray grad_array);

};
```

Indicate the number of parameters that the bias needs per catalog

The power law bias function

```
struct PowerLaw {
 const auto numParams = 1;

  double alpha;
  double nmean;
  SimpleAdaptor selection_adaptor;

  void setup_default(BiasParameters& params) {
    params[0] = 1.5;
  }

  void prepare(ForwardModel& fwd_model, const FinalDensityArray& final_density,
    double const _nmean, const BiasParameters& params, MetaSelect _select = MetaSelect())

  void cleanup() {}

  static bool check_bias_constraints(Array&& a) {
    return a[0] > 0 && a[0] < 5; // alpha can take any value
  }

  auto compute_density(const FinalDensityArray& array);

  auto apply_adjoint_gradient(const FinalDensityArray& array,
    TupleGradientLikelihoodArray grad_array);

};
```

Indicate the number of parameters that the bias needs per catalog

Your internal bias parameters

Compatible selection function adaptor (TBI and TBD)

Put a sensible default value for initializing the MC

The power law bias function

```
struct PowerLaw {
 const auto numParams = 1;

  double alpha;
  double nmean;
  SimpleAdaptor selection_adaptor;

  void setup_default(BiasParameters& params) {
    params[0] = 1.5;
  }

  void prepare(ForwardModel& fwd_model, const FinalDensityArray& final_density,
    double const _nmean, const BiasParameters& params, MetaSelect _select = MetaSelect())
  void cleanup() {}

  static bool check_bias_constraints(Array&& a) {
    return a[0] > 0 && a[0] < 5; // alpha can take any value
  }

  auto compute_density(const FinalDensityArray& array);
  auto apply_adjoint_gradient(const FinalDensityArray& array,
    TupleGradientLikelihoodArray grad_array);

};
```

Prepare to compute the biased density (e.g. FFT ...)

Cleanup the preparation

Enforce constraints on the parameters (positivity, ordering, etc)

Compute lazily the biased density

Apply the adjoint gradient matrix operator

The power law bias function

```
struct PowerLaw {
 const auto numParams = 1;

  double alpha;
  double nmean;
  SimpleAdaptor selection_adaptor;

  void setup_default(BiasParameters& params) {
    params[0] = 1.5;
  }

  void prepare(ForwardModel& fwd_model, const FinalDensityArray& final_density,
    double const _nmean, const BiasParameters& params, MetaSelect _select = MetaSelect())
  void cleanup() {}

  static bool check_bias_constraints(Array&& a) {
    return a[0] > 0 && a[0] < 5; // alpha can take any value
  }

  auto compute_density(const FinalDensityArray& array);
  auto apply_adjoint_gradient(const FinalDensityArray& array,
    TupleGradientLikelihoodArray grad_array);

};
```

Indicate the number of parameters that the bias needs per catalog

Your internal bias parameters

Compatible selection function adaptor (TBI and TBD)

Put a sensible default value for initializing the MC

Prepare to compute the biased density (e.g. FFT ...)

Cleanup the preparation

Enforce constraints on the parameters (positivity, ordering, etc)

Compute lazily the biased density

Apply the adjoint gradient matrix operator

The lazy density evaluator

```
template<typename FinalDensityArray>
        inline auto compute_density(const FinalDensityArray& array)

        ->decltype(
              std::make_tuple(
               b_va_fused<double>(
                  boost::bind(density_lambda, nmean, alpha, boost::placeholders::_1),
                  array)
               )
             )
```

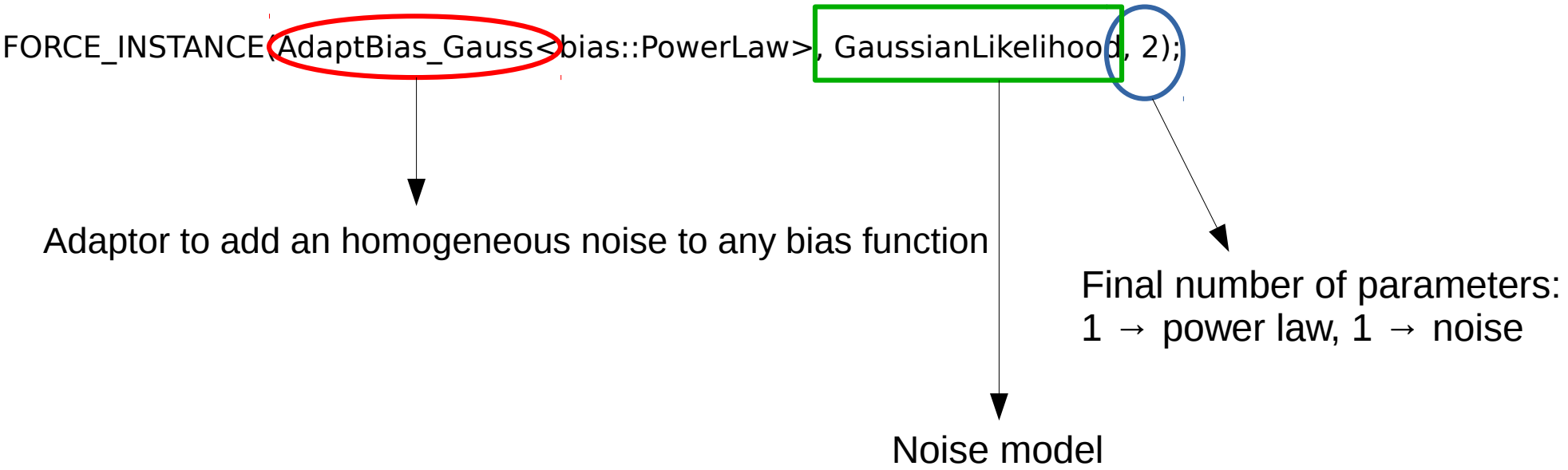
Required by C++11, not by C++14

```
{
  return std::make_tuple(
    b_va_fused<double>(
      boost::bind(density_lambda, nmean, alpha, boost::placeholders::_1),
      array)
    );
}
```

```
static inline double density_lambda(double nmean, double alpha, double v) {
  return nmean * std::pow(1+EPSILON_VOIDS+v, alpha);
}
```

Reduce compilation time: force implementation

Implementation is forced in libLSS/samplers/generic/impl_generic.cpp



Attaching to the main loop

There is a generic function that handles all the problems of generating meta sampler.

In hades_bundle.hpp you can just add a block like this one:

```
} else if (lh_type == "GAUSSIAN_POWERLAW_BIAS") {
  bundle.borg_generic = create_generic_bundle<
        AdaptBias_Gauss<bias::PowerLaw>,
        GaussianLikelihood
    > (bundle.comm, system_params, hmc, nmean, bias, vobs);
```


Same block as in impl_generic

AND THAT'S ALL FOLKS! You can start with Noop for a fresh bias.

Exercise: implement the ManyPower model

$$\rho_g = \sum_{i,j=0}^{N_{\max}} \delta^{i+j} A_{i,j}$$

with $A > 0$ the bias function becomes positive

Sufficient: $A = LL^T$, $L = \begin{pmatrix} a_{00} & 0 & \dots & \dots \\ a_{10} & a_{11} & 0 & \dots \\ \vdots & \ddots & \ddots & 0 \\ a_{n0} & \dots & \dots & a_{nn} \end{pmatrix}$

$$\frac{(N+1)(N+2)}{2}$$

parameters

$$A = \begin{pmatrix} a_{00}^2 & a_{00}a_{11} & \dots & a_{n0}a_{nn} \\ \vdots & a_{10}^2 + a_{11}^2 & \dots & a_{10}a_{n0} + a_{11}a_{n1} \\ \vdots & \ddots & \ddots & \vdots \\ \dots & \dots & \dots & \sum_{j=1}^{N_{\max}} a_{nj}^2 \end{pmatrix}$$