{{lastupdated_at}} by {{lastupdated_by}}

GModel

Pre computations

Model functions will be called very often during a typical GammaLib analysis, hence evaluation of models need to be as efficient as possible. Here, pre computations are key. When possible, models should have a cache of precomputed values that will be used until parameters change. Note here that a check is much quicker than the evaluation of, for example, and exponential function. Thus we can afford some checks of this speeds up computations at the end.

Model locking is a possibility, that should be investigated. When a model is locked, it can be assumed that model parameters will not be modified. Having this capability will allow an efficient usage of pre computation. Model locking can for example be used in likelihood computations, and can help to speed up things considerably.

Two methods need to be implemented for each model (and probably model component): lock() and unlock(). When the lock() method has been called, and until a call of unlock(), the model can assume that parameters will not be changed. This allows to perform pre-computations on locking, and to use cached values until the model is unlocked. lock() can also call methods to perform pre computations.

Care has to be taken for numerical derivative computation, which will change parameters. Here we have to make sure that the model is unlocked before the derivative computation starts, and locked afterwards. If this is done too often, the gain of pre computation may be lost.le

Parameter linking

It should be possible to link different parameters together. For this purpose, a link reference could be used, and by specifying this reference one could signal the GModels class that two or more parameters are in fact identical. The effective number of parameters will then be reduced.

Parameter functions

Instead of specifying a parameter one may want to specify a function of parameters. The normal GModelPar would then be a simple case of the identity relation. Parameter functions could then allow to have one model scaling with s and another with 1-s. It also would allow to have for example an energy dependent parameter (not sure that this is really the same case). A parameter function may have other parameters or parameter functions as members. This will allow to implement functional relations of arbitrary complexity.

We may implement an abstract class GModelFunction that provides the interface for a parameter function. GModelPar would then be derived from this class as the implementation of the identiy function. A model may then have GModelPar members and GModelFunction members. Here an example:

```
GModelPar      m_ra;
GModelPar      m_dec;
GModelFunction* m_radius;
```

This allows for example the implementation of an energy dependent radius. The model would access all members as they were simple parameters using the value method.
The GModelFunction should also have a gradient method that allows setting the gradient of the function parameters. The model class will set the gradient with respect to the function value, and the GModelFunction will compute the gradient of the function value with respect to the parameters. The GModelFunction::gradient method will multiply both for setting the parameter gradients.
The members of GModelFunction are either GModelPar parameters or GModelFunction classes. Functions can thus be nested, allowing the building of arbitrarily complex models.
Here an example of a model XML interface using functions:

```
<parameter name="RA" scale="1" ... />
<parameter name="DEC" scale="1" ... />
<function name="Radius" type="EnergyPowerLaw">
```

```
  <parameter name="Prefactor" scale="1" ... />
  <parameter name="Pivot" scale="1" ... />
  <function name="Index" type="sqrt">
    <parameter name="Index" scale="1" ... />
  </function>
</function>
```

This model has 6 parameters of which 3 are used in 2 functions to compute the radius of the model. This example illustrates how functions can be used to make arbitrary computations.