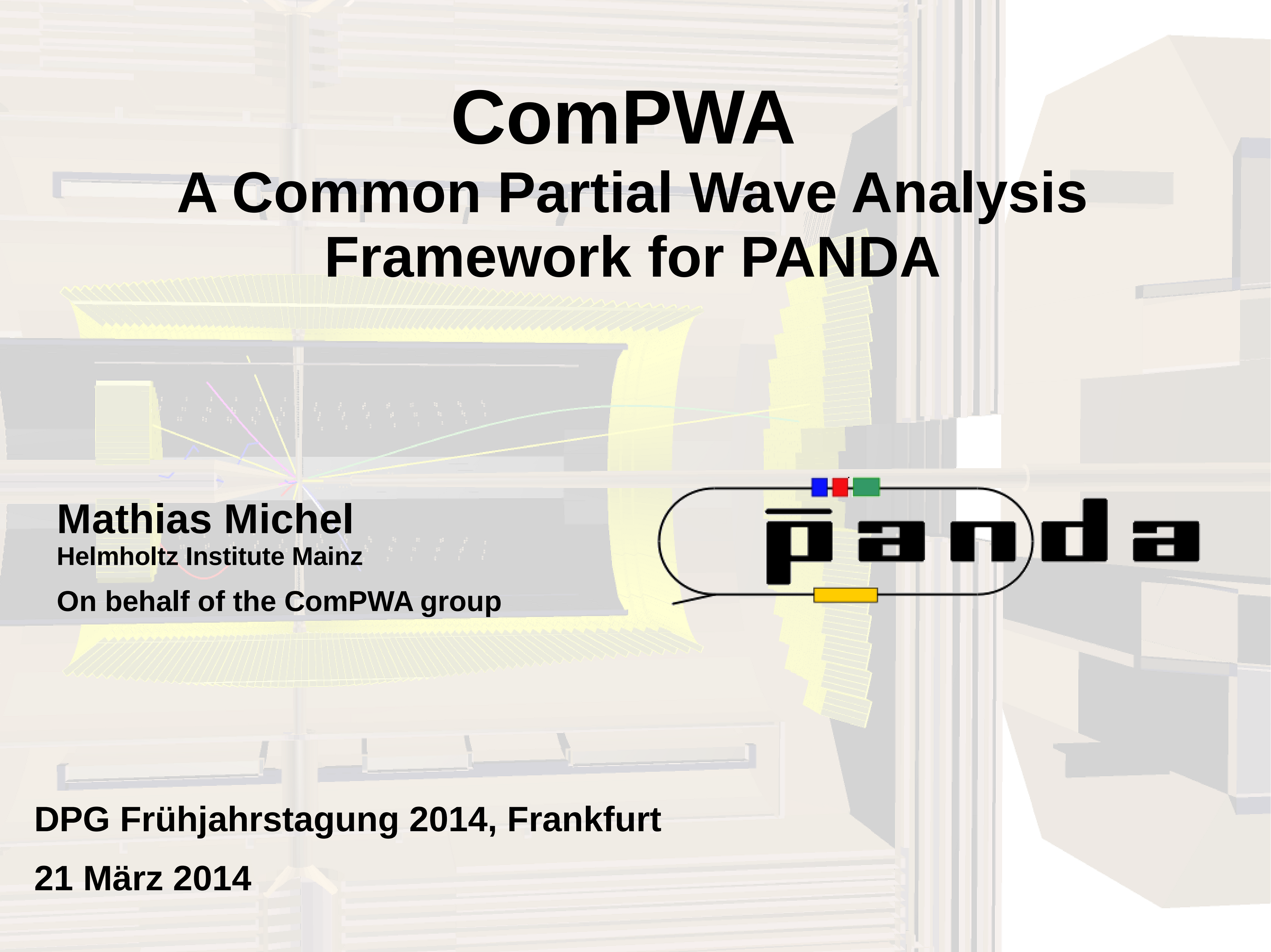


ComPWA

A Common Partial Wave Analysis Framework for PANDA

Mathias Michel

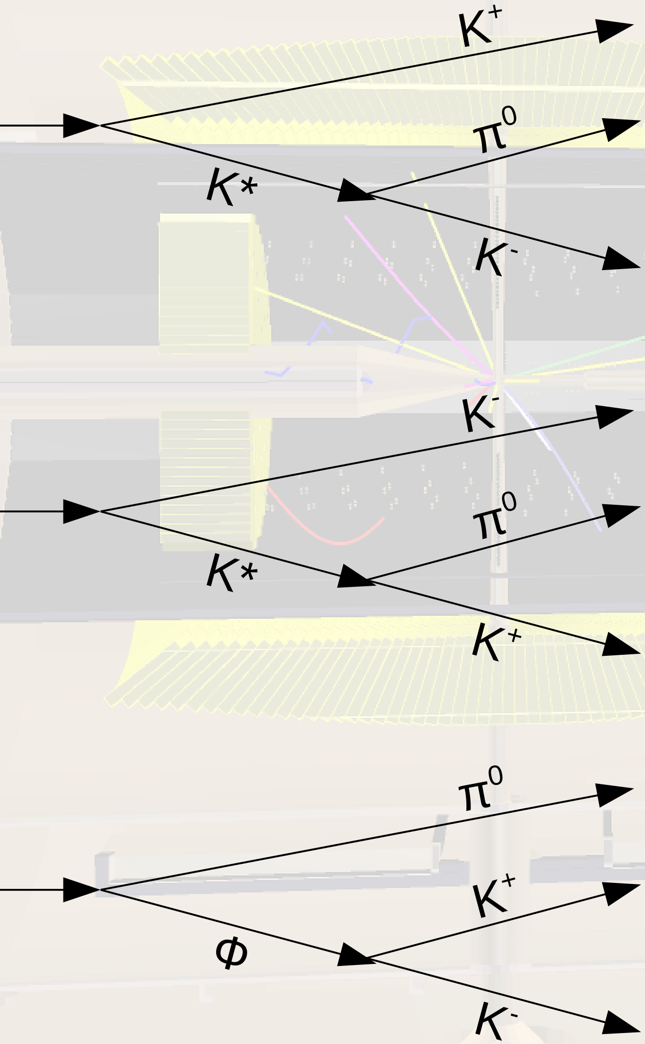
Helmholtz Institute Mainz

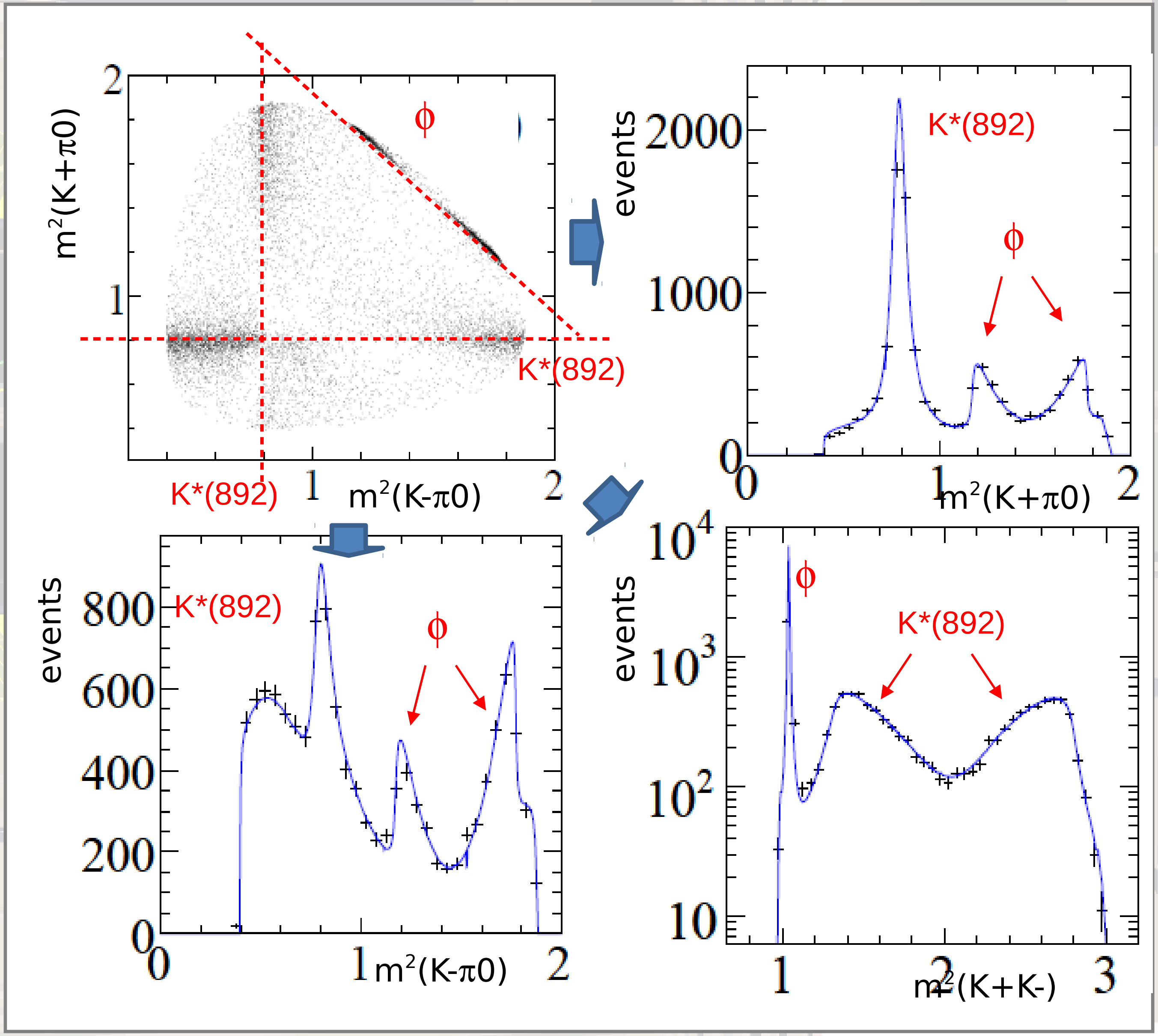
On behalf of the ComPWA group

DPG Frühjahrstagung 2014, Frankfurt

21 März 2014

PWA


K+
π0
K*
K-
K-
π0
K*
K+
π0
K+
ϕ
K-


m²(K+π0)
2
1
ϕ
K*(892)
K*(892)
1
m²(K-π0)
2
events
2000
1000
0
0
1
m²(K+π0)
2
K*(892)
ϕ
events
800
600
400
200
0
0
1
m²(K-π0)
2
K*(892)
ϕ
events
10⁴
10³
10²
10
1
2
m²(K+K-)
3
ϕ
K*(892)

ComPWA Challenges

Example: $\bar{\text{P}}$ANDA

broad physics program
→ various models needed

$\bar{p}p$ initial state at √s 2 – 5 GeV
→ high initial spin (≈ up to 6-7)
→ many possible waves
→ many parameters

large number of events
→ parallelization needed

detector effects
→ distorted phasespace
→ accaptence

coupled channels
→ different efficiencies

quality assurance

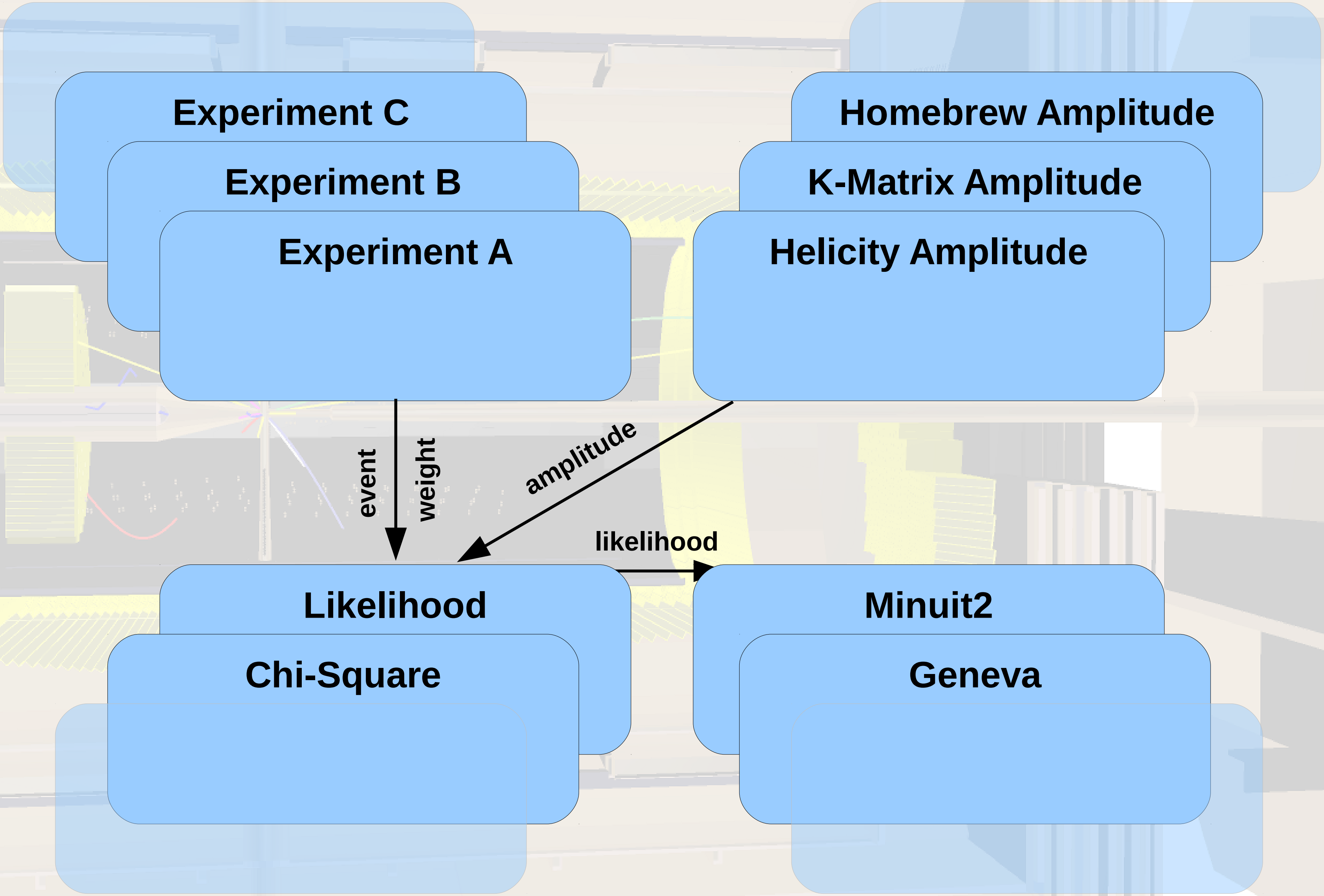
Why a *common* framework?

PWA tools on the market are specialised
→ not extendible

PANDA is still years from data taking
→ start now and we could have a well tested and reliable software ready

comparison of results from different experiments possible

ComPWA Framework



ComPWA Framework

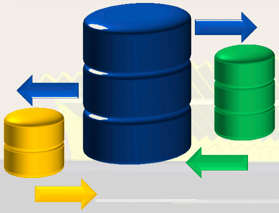
Data & MC

represents measurements

local and global values

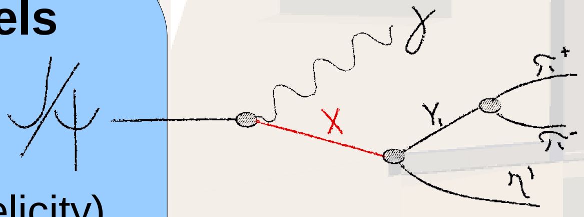
multiple experiments (at once)

caching



Physics & Models

calculates amplitudes



various formalisms (helicity)

various models (isobar)

simple ways to add new modules (wrapper)

event

weight

amplitude

likelihood

Controls & Book Keeping

user interface & run manager

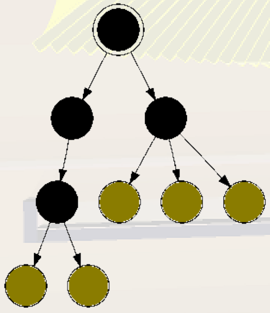
Estimators

calculates discrepancy from model to data

functiontree

combined fits and re-fits

documentation of procedure



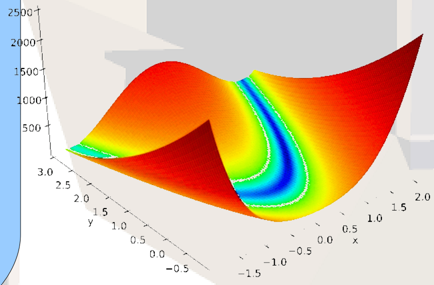
Optimizer

varies model parameter

interface to **external** libraries

various algorithms (gradient decent, genetic, swarm)

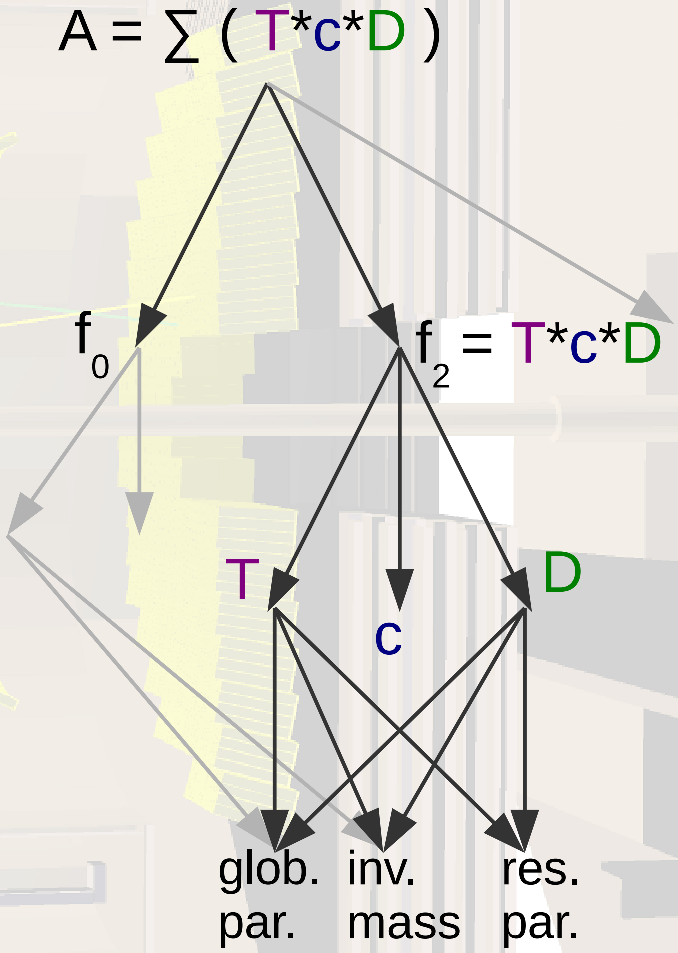
flexible strategies



FunctionTree

$$I = \left| \sum_n T_n c_n D_n \right|^2$$

T = Breit-Wigner Function
D = D-Wigner Function
c = Complex Factor



A = ∑ (T*c*D)

f_0

f_2 = T*c*D

T

c

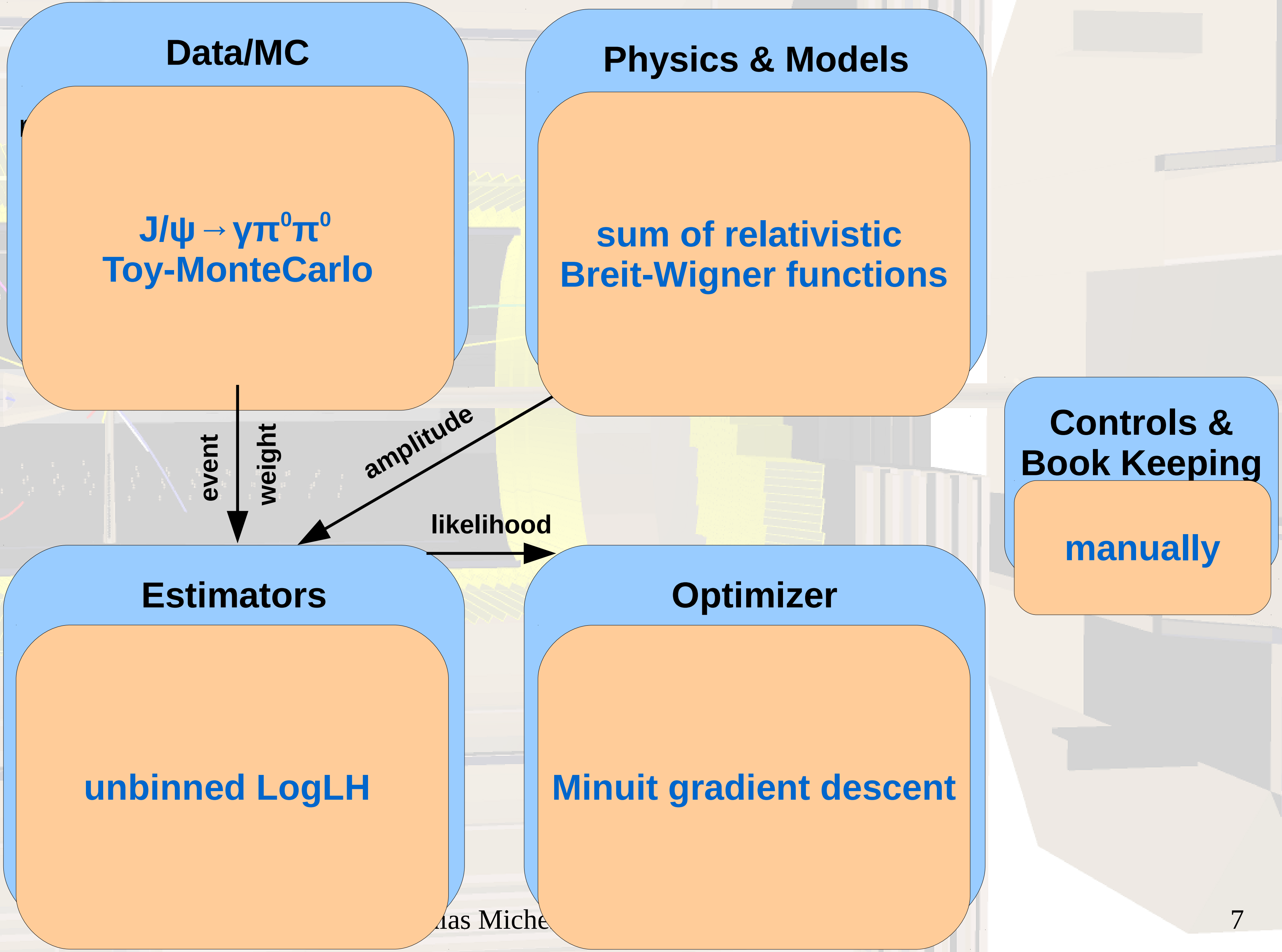
D

glob. par.

inv. mass

res. par.

Test Environment: $J/\psi \rightarrow \gamma\pi^0\pi^0$



Data/MC

$J/\psi \rightarrow \gamma\pi^0\pi^0$
Toy-MonteCarlo

Physics & Models

sum of relativistic
Breit-Wigner functions

Controls & Book Keeping

manually

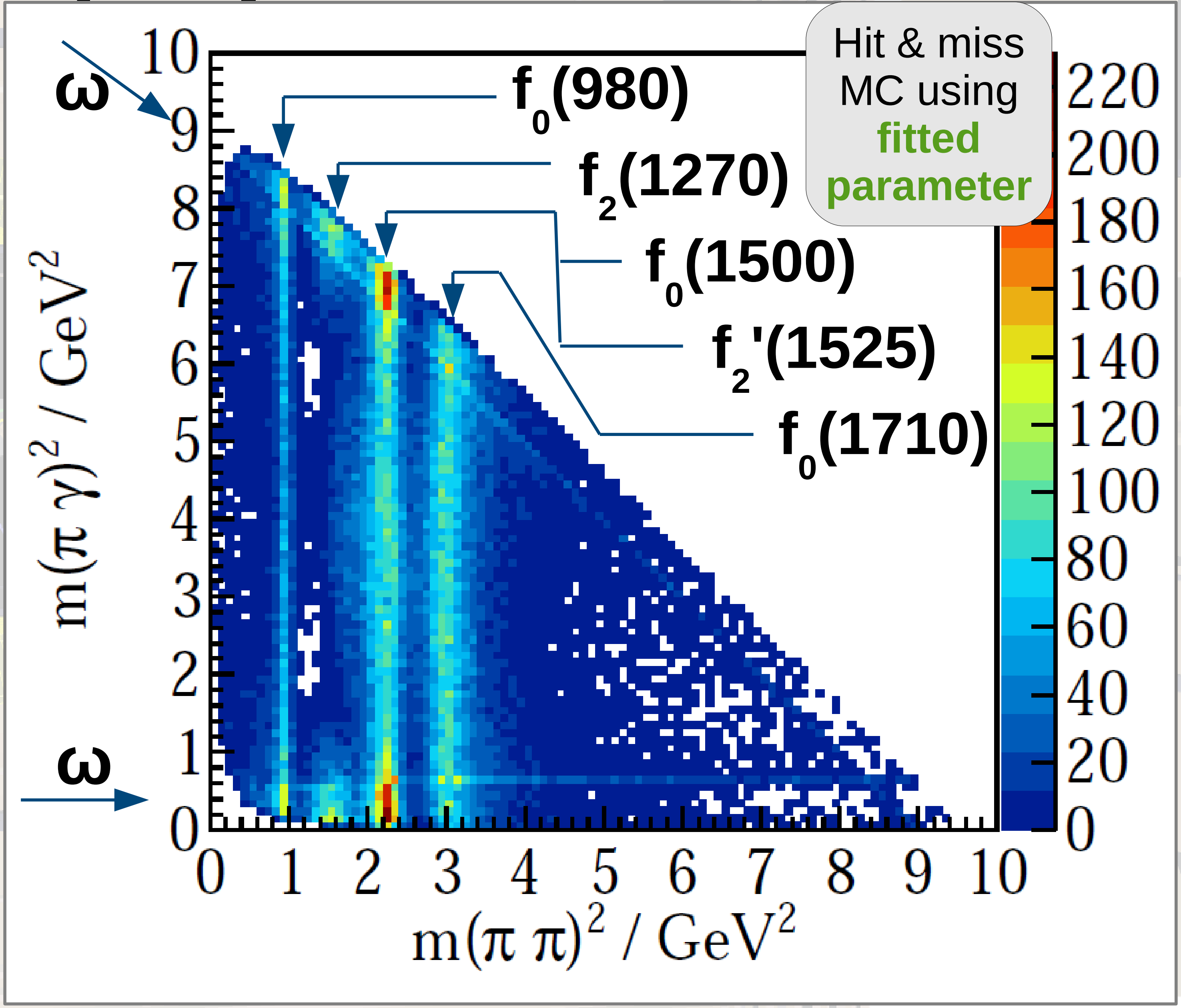
Estimators

unbinned LogLH

Optimizer

Minuit gradient descent

$J/\psi \rightarrow \gamma\pi^0\pi^0$ first fit



Toy-MC fit of f-resonance positions

startvalues: 0.95*ideal

reson.	ideal	result
$f_0(980)$	0.990	0.9900
$f_0(1500)$	1.505	1.5048
$f_0(1710)$	1.720	1.7205
$f_2(1270)$	1.274	1.2725
$f_2'(1525)$	1.525	1.5245

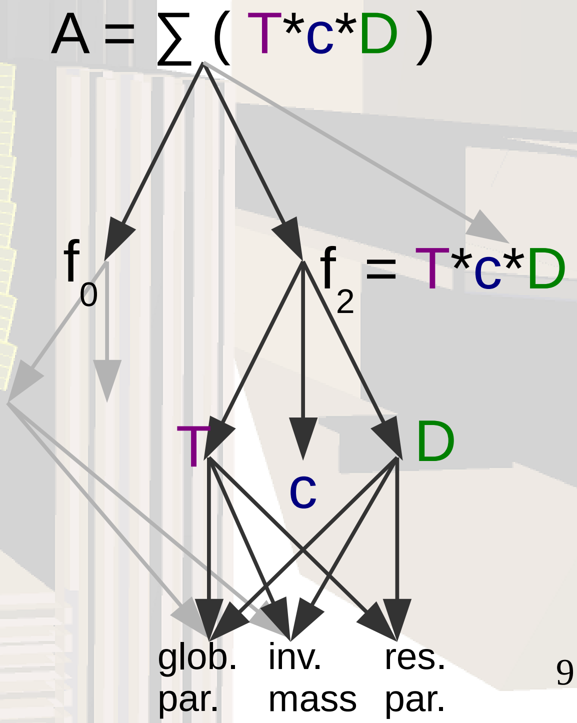
CPU-Time

Scenario (fit variables)	Avg. CPUTime per iteration without Tree	Avg. CPUTime per iteration with Tree	Speedup
1 Intensity	29.96 s	0.57 s	>50
5 Intensitie's	47.74 s	1.65 s	>25
5 BW-Width's	44.19 s	5.01 s	>8

100 000 Events, 7 resonances

each fit repeated:
- 10 times without tree
- 20 times with tree

- Speedup strongly dependent on use case
- Simple decay, simple model, few resonances → small tree
- Big trees → save memory by collapsing



Status & Outlook

Language and Dependencies
- C++11
- Boost
- Boost.Build

External Packages Used
- Root
- qft++
- Minuit2
- Geneva

Documentation
- Doxygen
- Doku Wiki

Version-Control
- Git

About Geneva

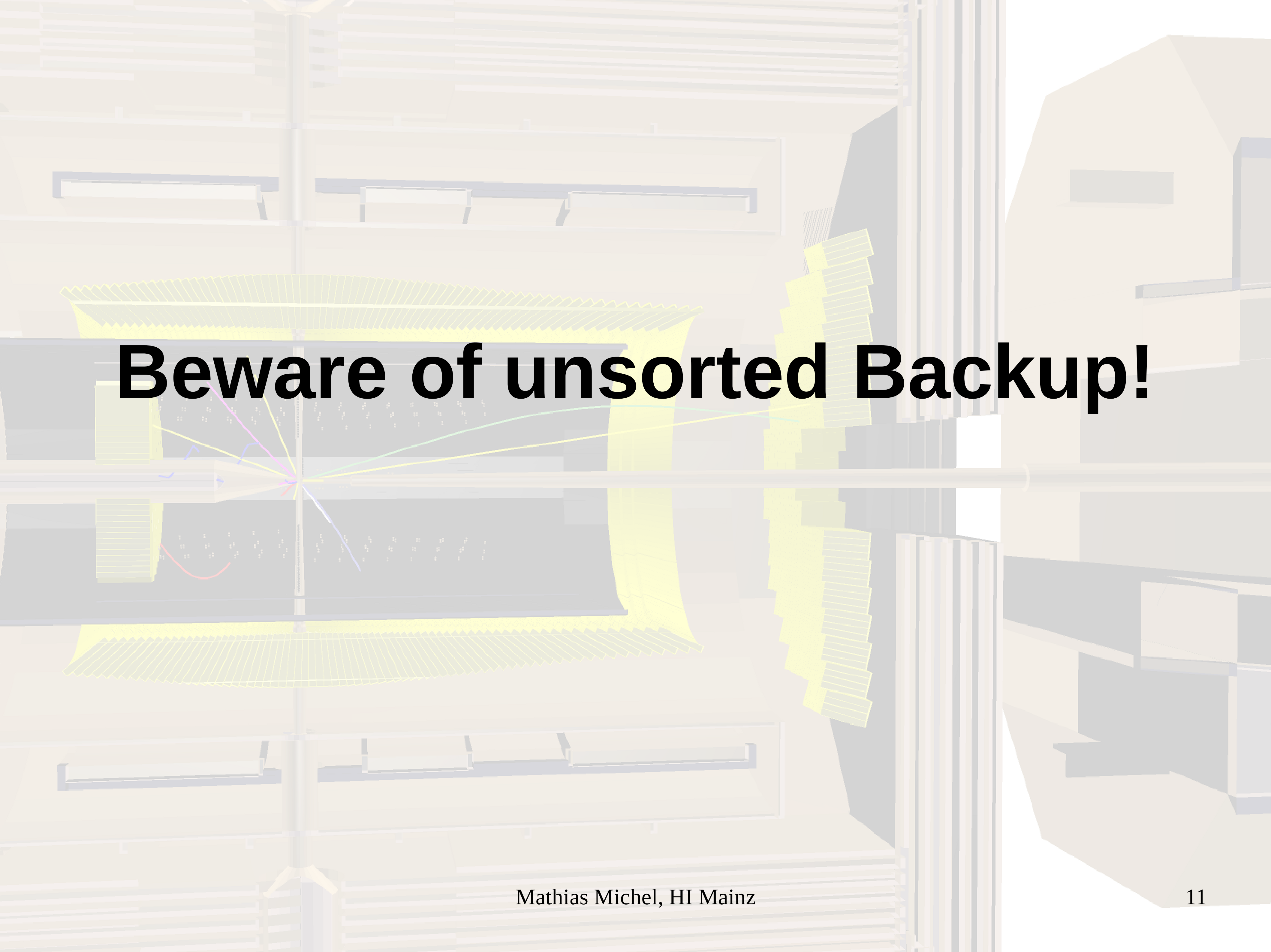
Geneva is available as Open Source software (AGPL v3) from **http://www.launchpad.net/geneva,** and is also supported commercially by Gemfony scientific **http://www.gemfony.eu**

This is work in progress!

Biggest ToDo's
- controls and configuration
- book keeping module
- more physics cases

Contact

github.com/ComPWA/ComPWA
michel@kph.uni-mainz.de



Beware of unsorted Backup!

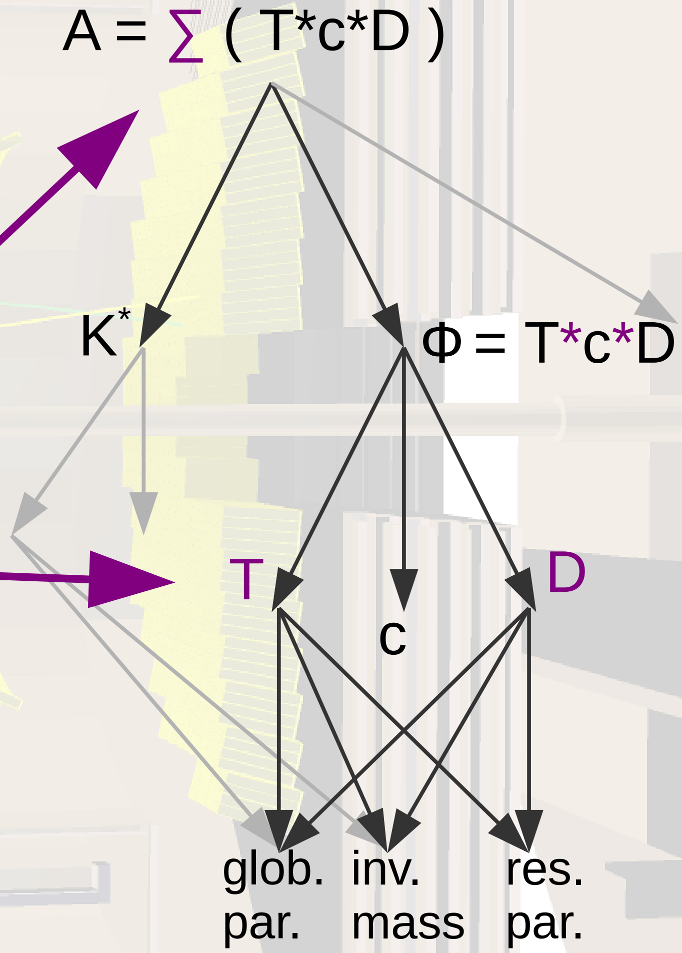
FunctionTree: Strategies

Node

List of parents
List of children
My parameter
Strategy

Strategy pattern

execute function
input:
parameter list (children)
output:
calculated parameter

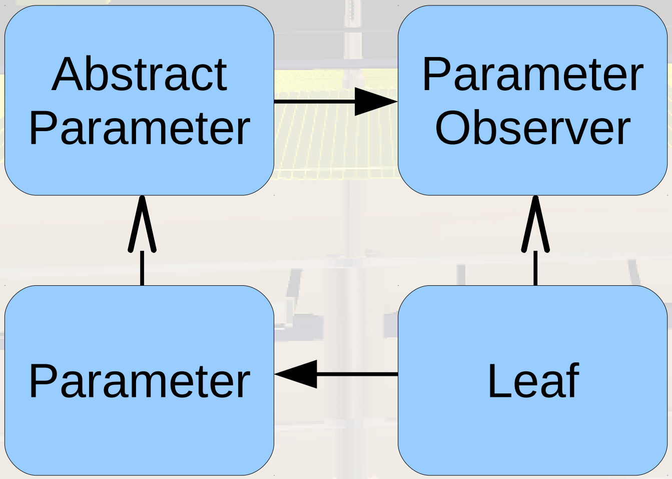


FunctionTree: Parameters

Leaf

List of parents
~~List of children~~
My Parameter
~~Strategy~~

Observer pattern

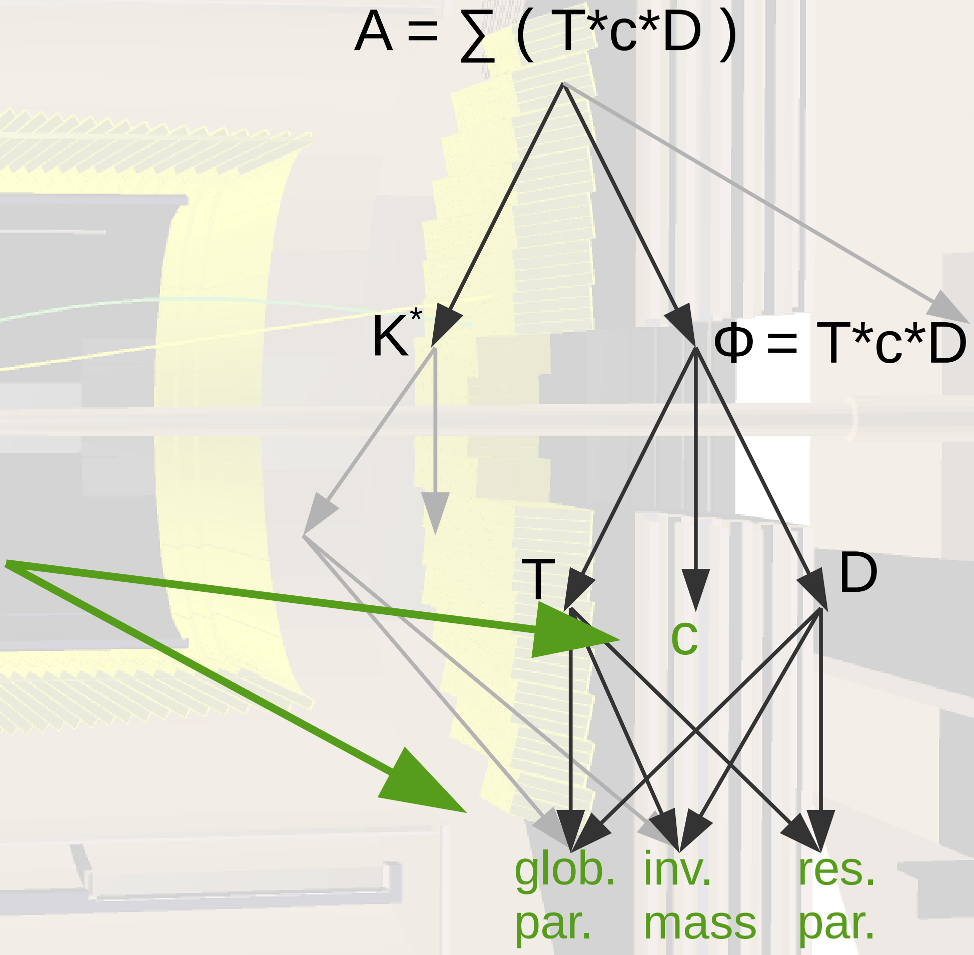


Abstract Parameter

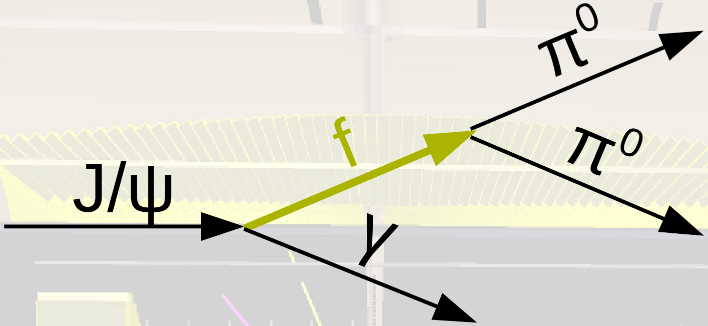
Parameter Observer

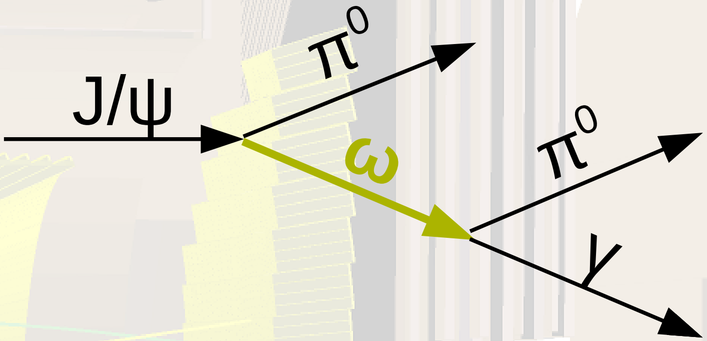
Parameter

Leaf



$J/\psi \rightarrow \gamma\pi^0\pi^0$ Model

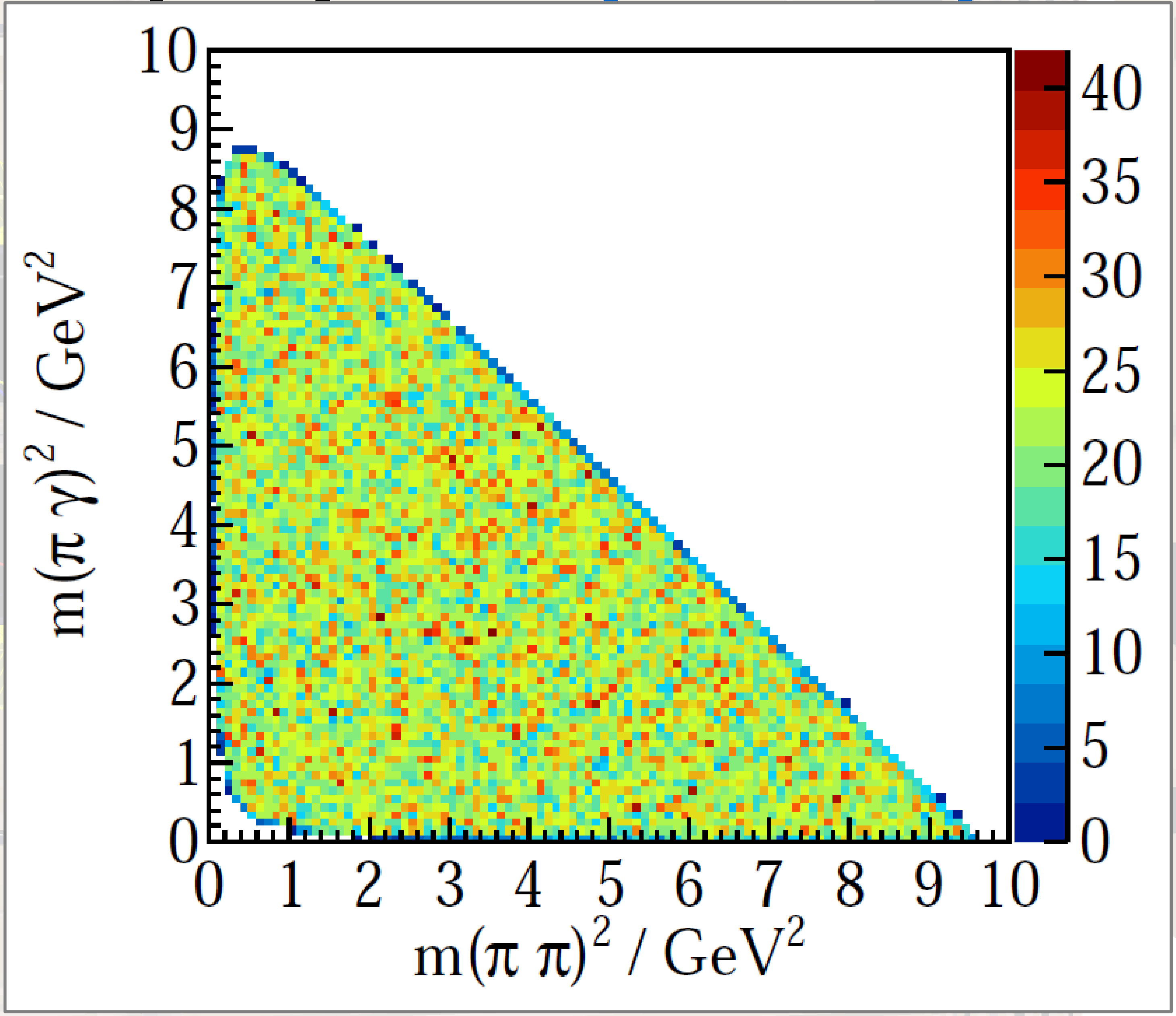




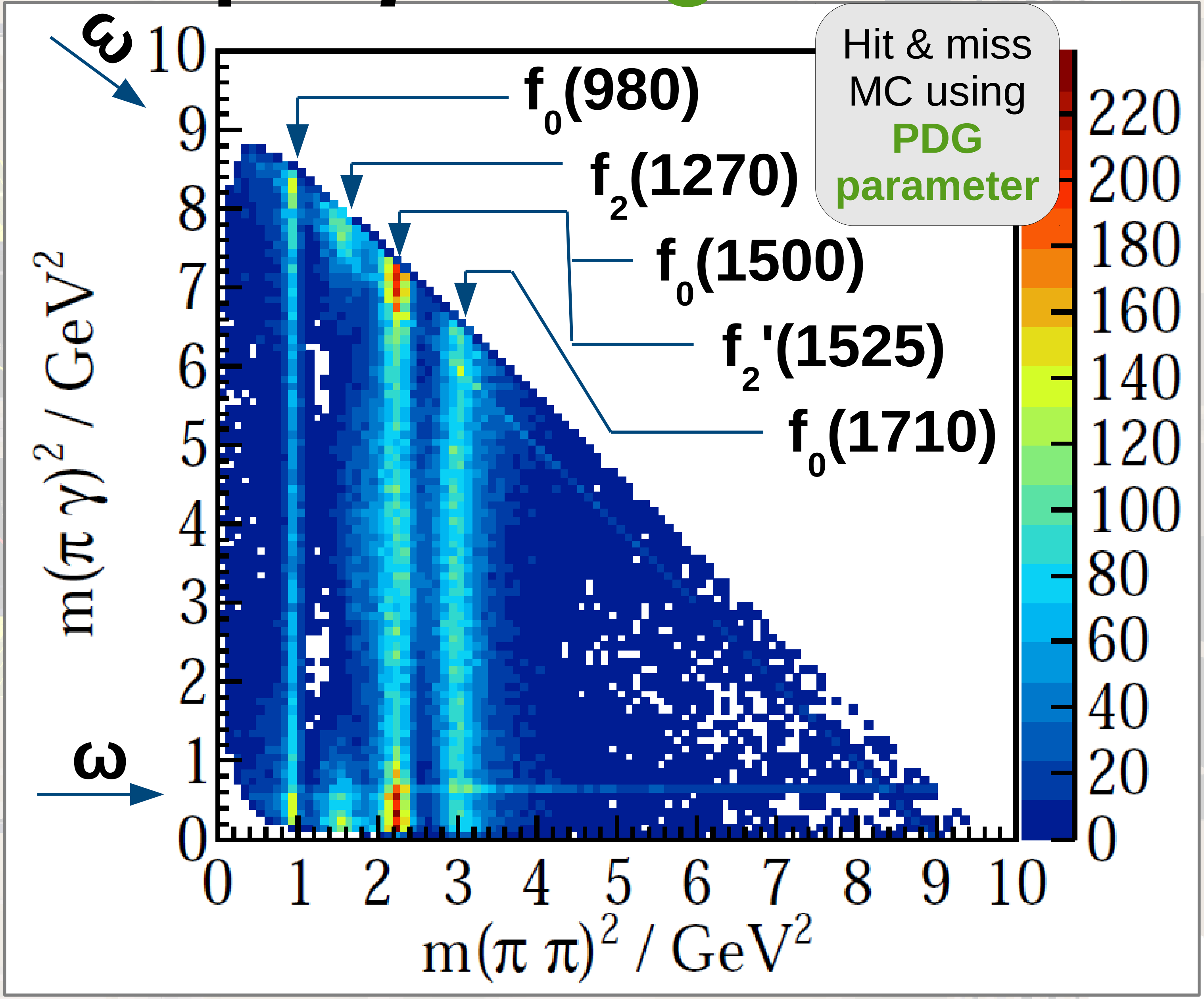
$$I=\left|\sum_n T_n r_n e^{i\varphi_n} D_n\right|^2$$

T = Breit-Wigner Function
D = D-Wigner Function
r = Strength of Resonance
φ = Phase of Resonance

$J/\psi \rightarrow \gamma\pi^0\pi^0$ phasespace

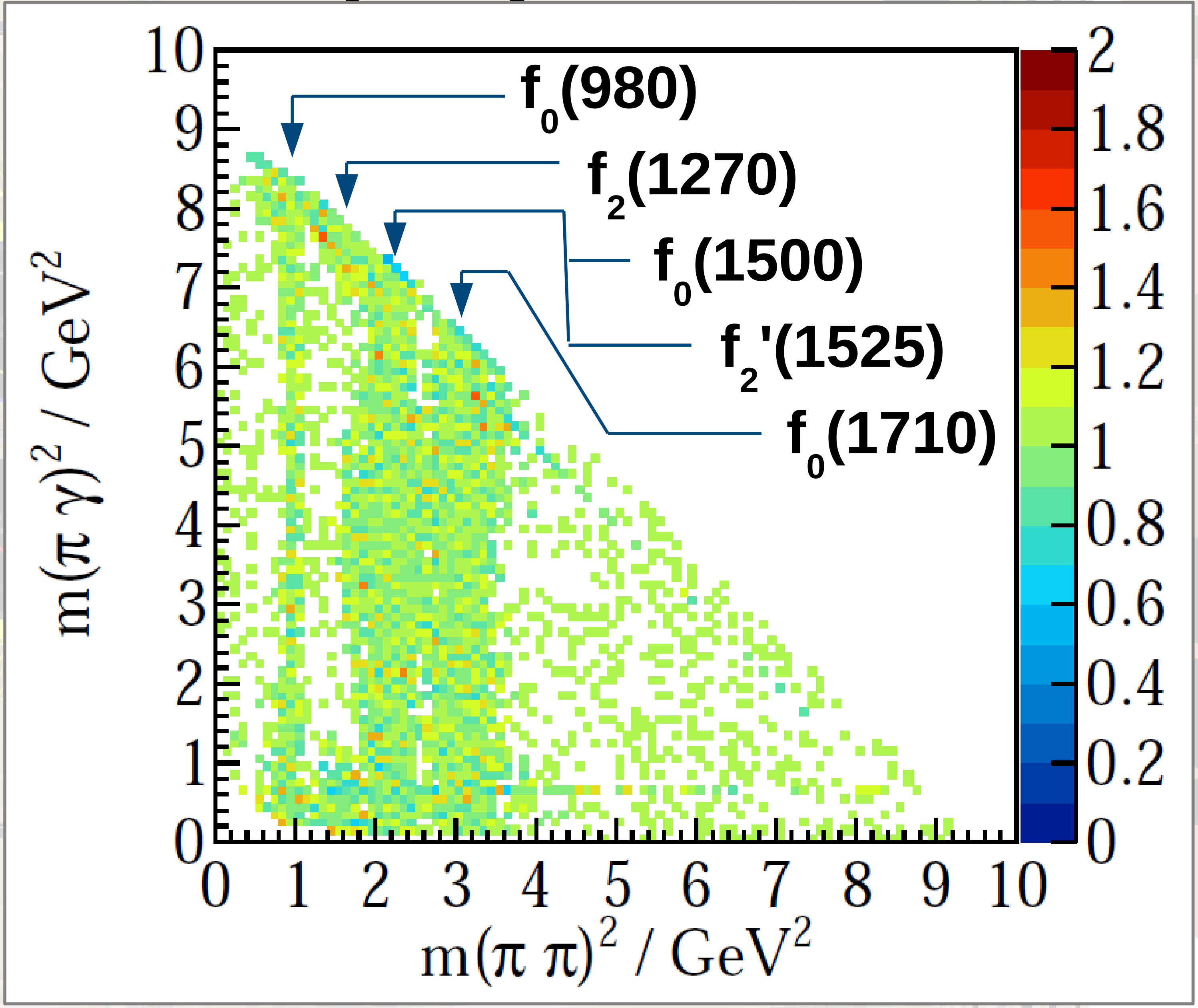


$J/\psi \to \gamma\pi^0\pi^0$ generated



Hit & miss MC using **PDG parameter**

$J/\psi \rightarrow \gamma\pi^0\pi^0$ ratio



Summary

Modules:

- Ascii & Root Data Reader
- 2 Particle Breit-Wigner Amplitude
- 3 Particle Breit-Wigner Sum Amplitude
- X^2 & logLH Estimator
- Minuit2 & Geneva Optimizer Interfaces

First Test Enviroments:

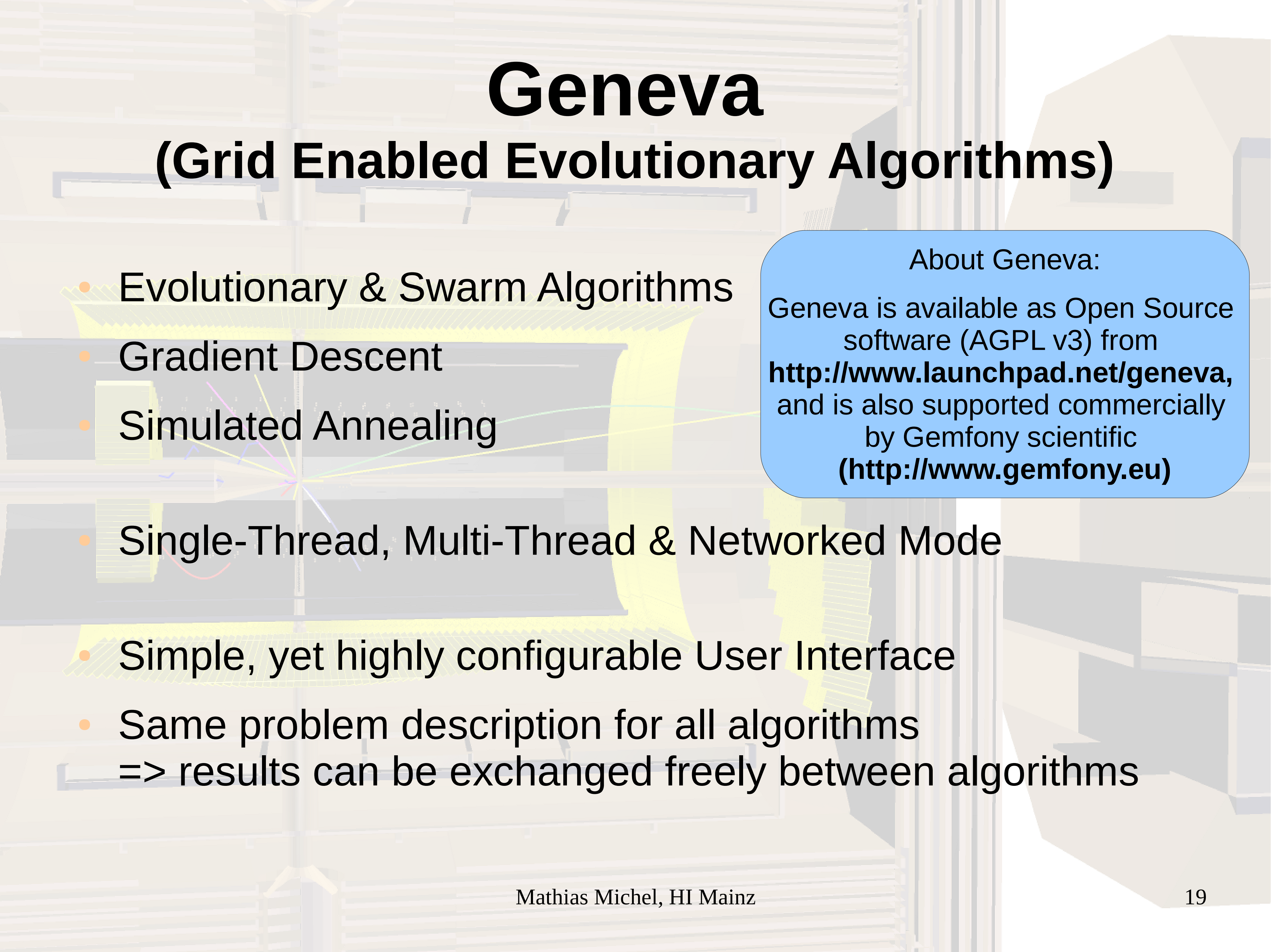
- Two Particle Breit-Wigner Full Fit
- Three Particle Breit-Wigner Sum Full Fit

Dictionary and FunctionTree:

- FunctionTree ready, needs testing
- Dictionary work in progress

ToDo:

- Documentation of fit
- Control-Module, configuration
- License
- ...



Geneva

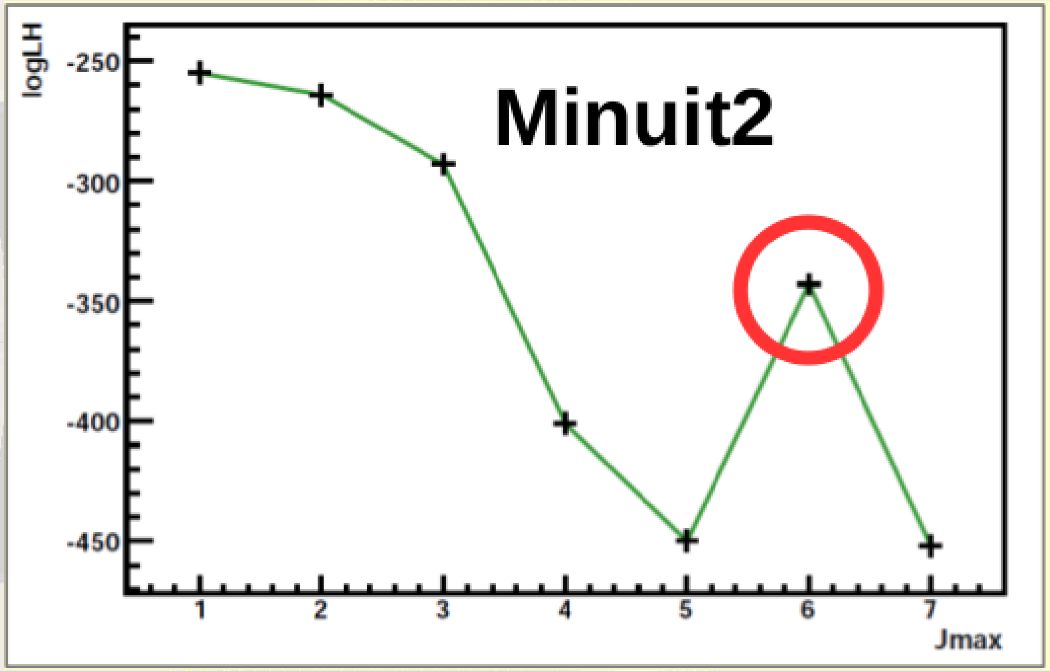
(Grid Enabled Evolutionary Algorithms)

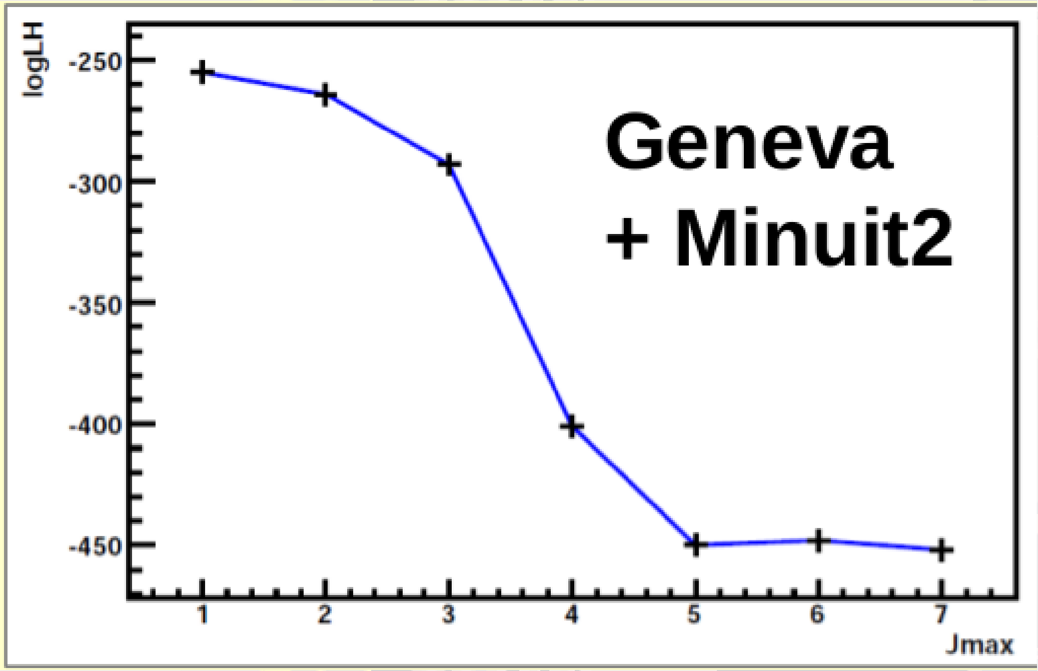
- Evolutionary & Swarm Algorithms
- Gradient Descent
- Simulated Annealing
- Single-Thread, Multi-Thread & Networked Mode
- Simple, yet highly configurable User Interface
- Same problem description for all algorithms
  => results can be exchanged freely between algorithms

About Geneva:

Geneva is available as Open Source software (AGPL v3) from **http://www.launchpad.net/geneva,** and is also supported commercially by Gemfony scientific **(http://www.gemfony.eu)**

Geneva
(Grid Enabled Evolutionary Algorithms)

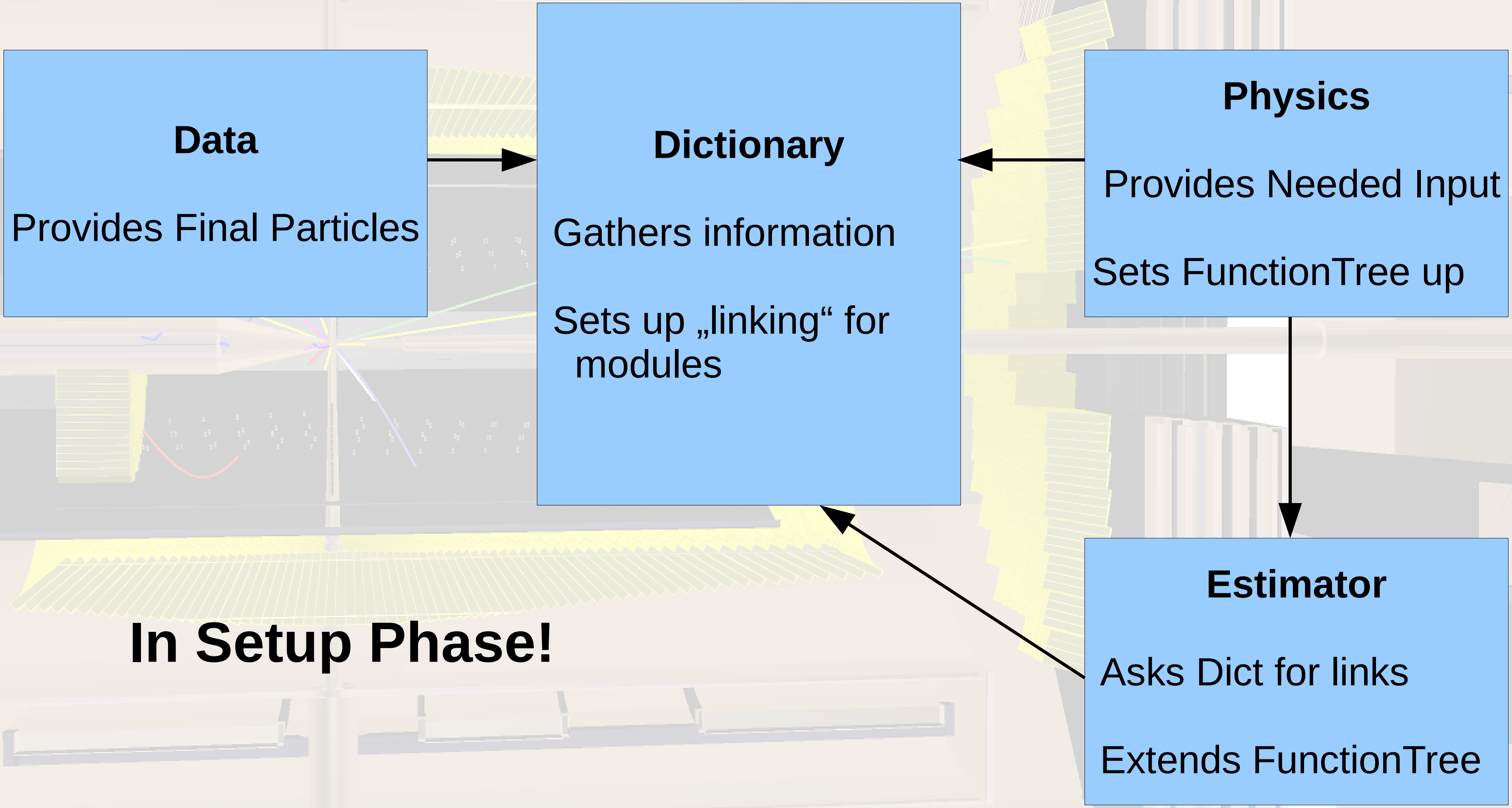




```
<name>f0_980</name>
<mass>0.99</mass>
<width>0.05</width>
<strength>1.</strength>
<phase>0.</phase>
<spin>0</spin>
<m>0</m>
<n>0</n>
<daugtherA>2</daugtherA>
<daugtherB>3</daugtherB>

<name>f0_1500</name>
<mass>1.505</mass>
<width>0.109</width>
<strength>1.</strength>
<phase>0.</phase>
<spin>0</spin>
<m>0</m>
<n>0</n>
<daugtherA>2</daugtherA>
<daugtherB>3</daugtherB>

<name>f0_1710</name>
<mass>1.72</mass>
<width>0.135</width>
<strength>1.</strength>
<phase>0.</phase>
<spin>0</spin>
<m>0</m>
<n>0</n>
<daugtherA>2</daugtherA>
<daugtherB>3</daugtherB>

<name>f2_1270</name>
<mass>1.274</mass>
<width>0.185</width>
<strength>1.</strength>
<phase>0.</phase>
<spin>2</spin>
<m>0</m>
<n>0</n>
<daugtherA>2</daugtherA>
<daugtherB>3</daugtherB>
```

```
<name>f2_1525</name>
<mass>1.525</mass>
<width>0.073</width>
<strength>1.</strength>
<phase>0.</phase>
<spin>2</spin>
<m>0</m>
<n>0</n>
<daugtherA>2</daugtherA>
<daugtherB>3</daugtherB>

<name>omega</name>
<mass>1.</mass>
<width>0.05</width>
<strength>1.</strength>
<phase>0.</phase>
<spin>0</spin>
<m>0</m>
<n>0</n>
<daugtherA>1</daugtherA>
<daugtherB>2</daugtherB>

<name>omega</name>
<mass>1.</mass>
<width>0.05</width>
<strength>1.</strength>
<phase>0.</phase>
<spin>0</spin>
<m>0</m>
<n>0</n>
<daugtherA>1</daugtherA>
<daugtherB>3</daugtherB>
```

Dictionary & FunctionTree



Data

Provides Final Particles

Dictionary

Gathers information

Sets up „linking“ for modules

Physics

Provides Needed Input

Sets FunctionTree up

Estimator

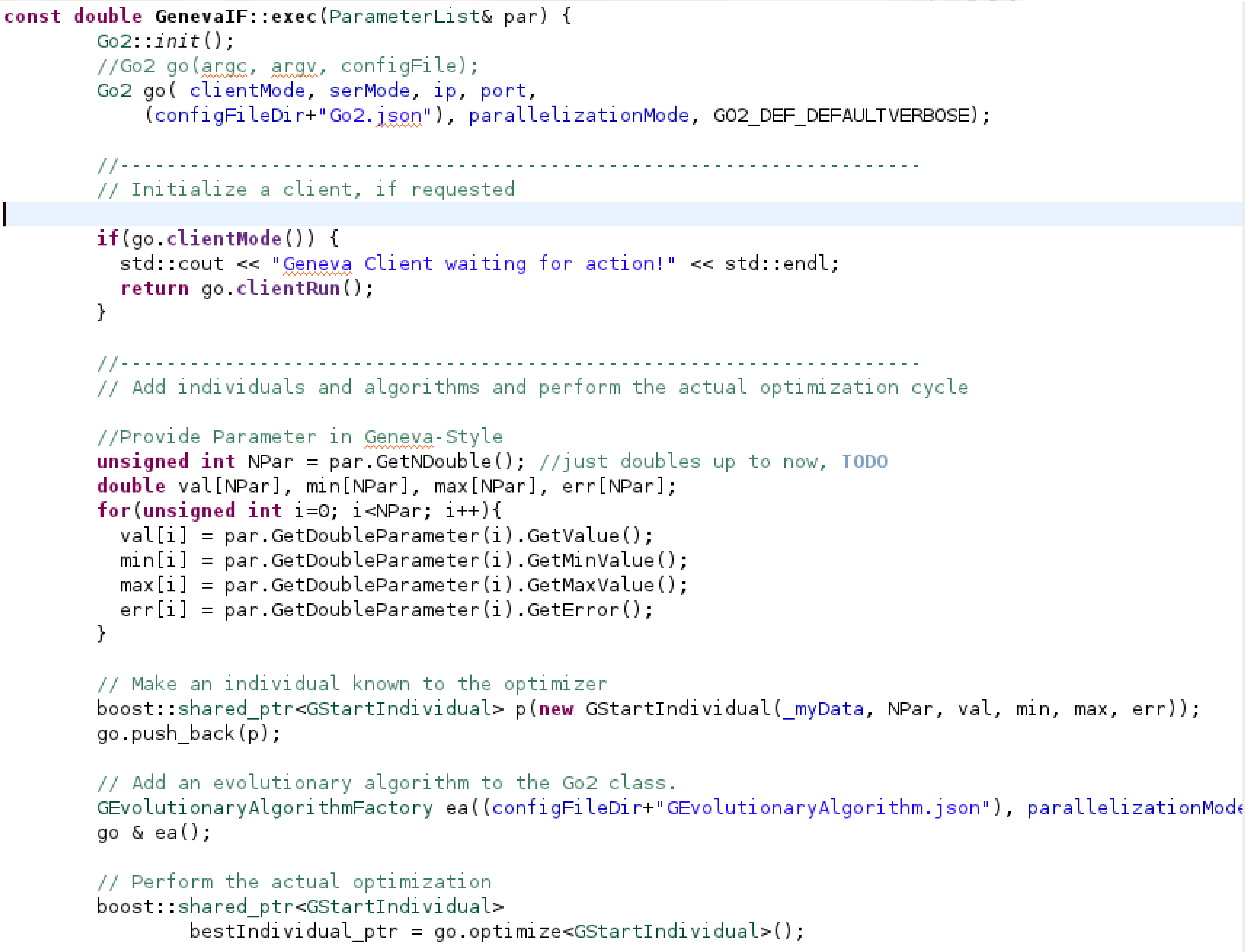
Asks Dict for links

Extends FunctionTree

In Setup Phase!

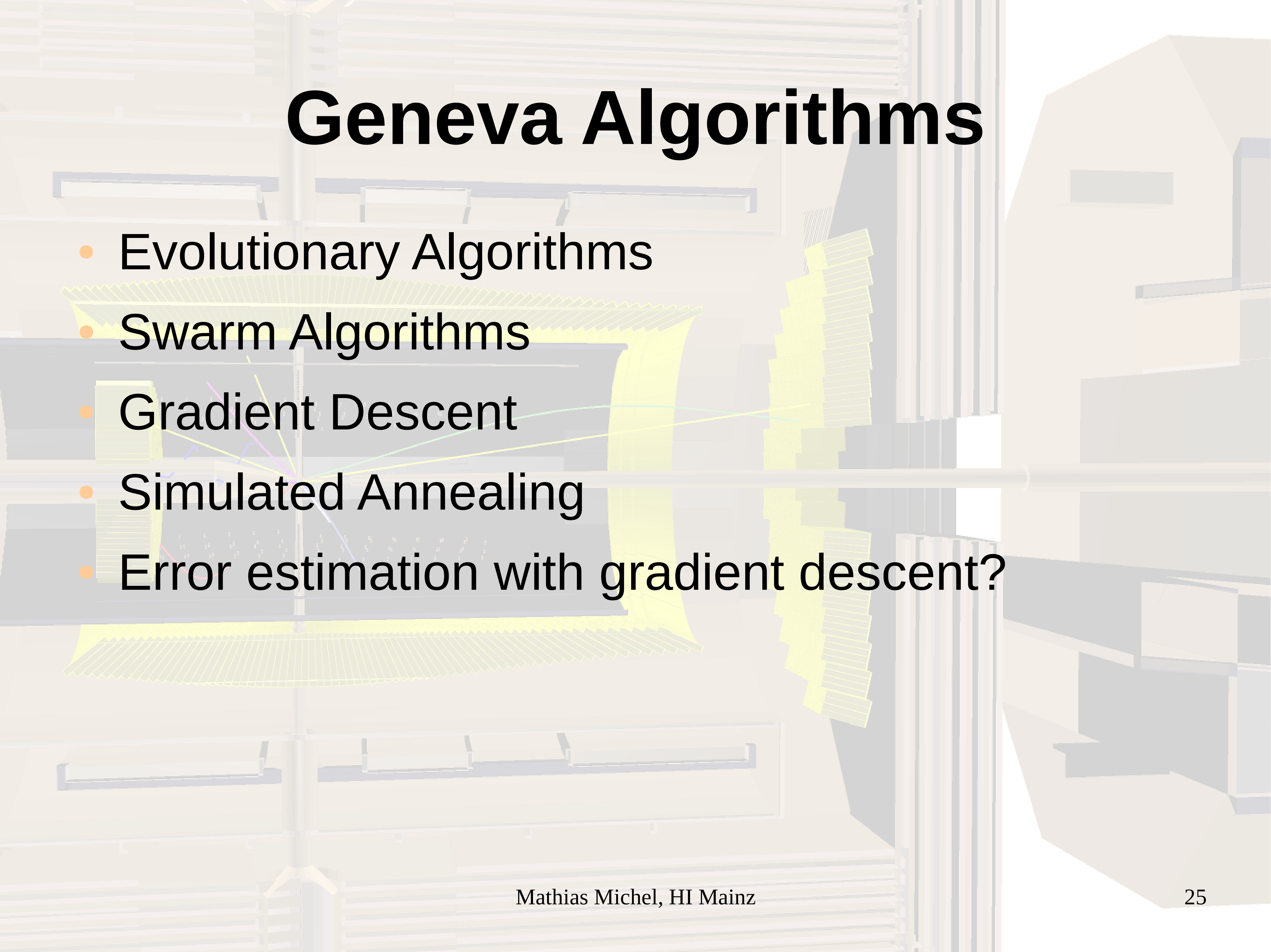
FunctionTree: Usage

```
//------------Setup Tree---------------------
myTree = std::shared_ptr<FunctionTree>(new FunctionTree());

//----Strategies needed
rbwStrat = std::shared_ptr<BreitWignerStrategy>(new BreitWignerStrategy());
angdStrat = std::shared_ptr<WignerDStrategy>(new WignerDStrategy());
multStrat = std::shared_ptr<MultAll>(new MultAll());
addStrat = std::shared_ptr<AddAll>(new AddAll());

//----Add Nodes
myTree->createHead("Amplitude", addStrat); //A=Sum{Resos}

//----Parameters needed
//unsigned int numReso = ini.getResonances().size();
for(std::vector<Resonance>::iterator reso=ini.getResonances().begin(); reso!=ini.getResonances().end(); reso++){
  Resonance tmp = (*reso);
  //setup RooVars
  mr.push_back( std::shared_ptr<DoubleParameter> (new DoubleParameter(("m_{"+tmp.m_name+"}"), tmp.m_mass, tmp.m_mass_min, tmp.m_mass_max) ) );
          ⋮

  //----Add Nodes
     unsigned int last = mr.size()-1;
  myTree->createNode("Reso_"+tmp.m_name, multStrat, "Amplitude"); //Reso=BW*c*AD
  myTree->createNode("RelBW_"+tmp.m_name, rbwStrat, "Reso_"+tmp.m_name); //BW
  myTree->createLeaf("Intens_"+tmp.m_name, rr[last], "Reso_"+tmp.m_name); //c
  myTree->createNode("AngD_"+tmp.m_name, angdStrat, "Reso_"+tmp.m_name); //AD
  //BW Par
  myTree->createLeaf("m0_"+tmp.m_name, mr[last], "RelBW_"+tmp.m_name); //m0
          ⋮
```



```cpp
const double GenevaIF::exec(ParameterList& par) {
        Go2::init();
        //Go2 go(argc, argv, configFile);
        Go2 go( clientMode, serMode, ip, port,
            (configFileDir+"Go2.json"), parallelizationMode, GO2_DEF_DEFAULTVERBOSE);

        //----------------------------------------------------------------------
        // Initialize a client, if requested

        if(go.clientMode()) {
          std::cout << "Geneva Client waiting for action!" << std::endl;
          return go.clientRun();
        }

        //----------------------------------------------------------------------
        // Add individuals and algorithms and perform the actual optimization cycle

        //Provide Parameter in Geneva-Style
        unsigned int NPar = par.GetNDouble(); //just doubles up to now, TODO
        double val[NPar], min[NPar], max[NPar], err[NPar];
        for(unsigned int i=0; i<NPar; i++){
          val[i] = par.GetDoubleParameter(i).GetValue();
          min[i] = par.GetDoubleParameter(i).GetMinValue();
          max[i] = par.GetDoubleParameter(i).GetMaxValue();
          err[i] = par.GetDoubleParameter(i).GetError();
        }

        // Make an individual known to the optimizer
        boost::shared_ptr<GStartIndividual> p(new GStartIndividual(_myData, NPar, val, min, max, err));
        go.push_back(p);

        // Add an evolutionary algorithm to the Go2 class.
        GEvolutionaryAlgorithmFactory ea((configFileDir+"GEvolutionaryAlgorithm.json"), parallelizationMode
        go & ea();

        // Perform the actual optimization
        boost::shared_ptr<GStartIndividual>
                bestIndividual_ptr = go.optimize<GStartIndividual>();
```

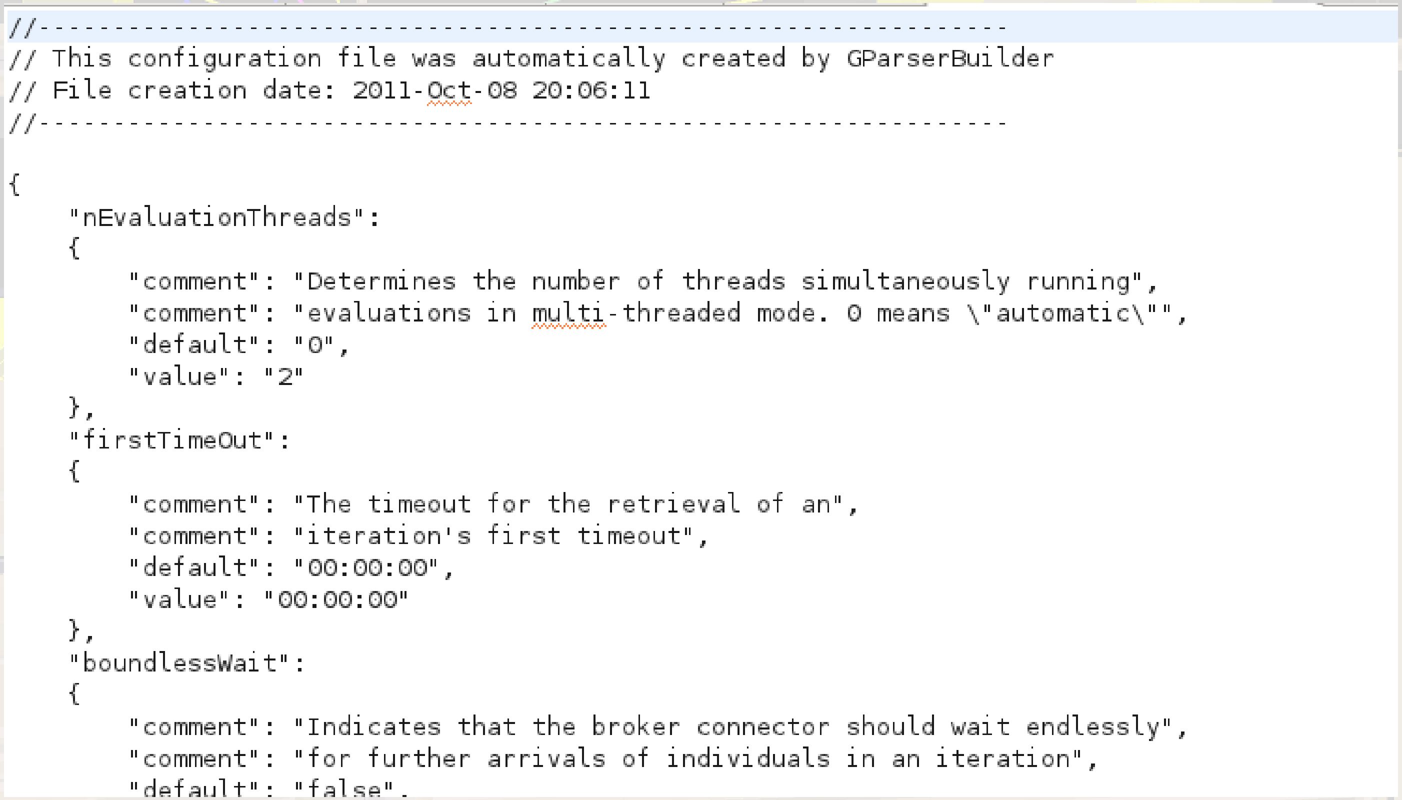
Geneva Algorithms



- Evolutionary Algorithms
- Swarm Algorithms
- Gradient Descent
- Simulated Annealing
- Error estimation with gradient descent?

Geneva User Interface

- Configuration: json files
- Go2 (connection settings)
- Algorithm (e.g. EA: genetic modification setup)



```
//-------------------------------------------------------------------
// This configuration file was automatically created by GParserBuilder
// File creation date: 2011-Oct-08 20:06:11
//-------------------------------------------------------------------

{
    "nEvaluationThreads":
    {
        "comment": "Determines the number of threads simultaneously running",
        "comment": "evaluations in multi-threaded mode. 0 means \"automatic\"",
        "default": "0",
        "value": "2"
    },
    "firstTimeOut":
    {
        "comment": "The timeout for the retrieval of an",
        "comment": "iteration's first timeout",
        "default": "00:00:00",
        "value": "00:00:00"
    },
    "boundlessWait":
    {
        "comment": "Indicates that the broker connector should wait endlessly",
        "comment": "for further arrivals of individuals in an iteration",
        "default": "false",
```

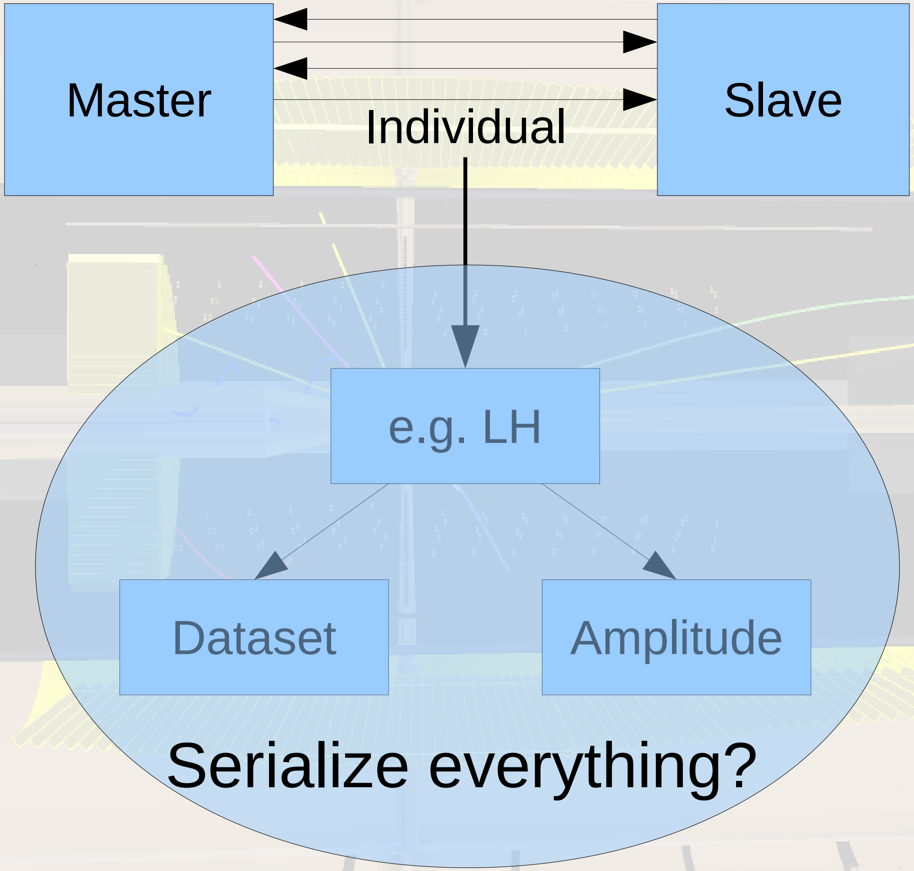
Geneva Networked

- Two kind of binaries: master and slave

1. Master sets problem up, waits for slaves
2. Slave ask master for problem-set, is registered
3. Slave performs optimization, sends result back
4. Master gathers results, sends new problems
5. Repeat point 3-4 until master solved problem

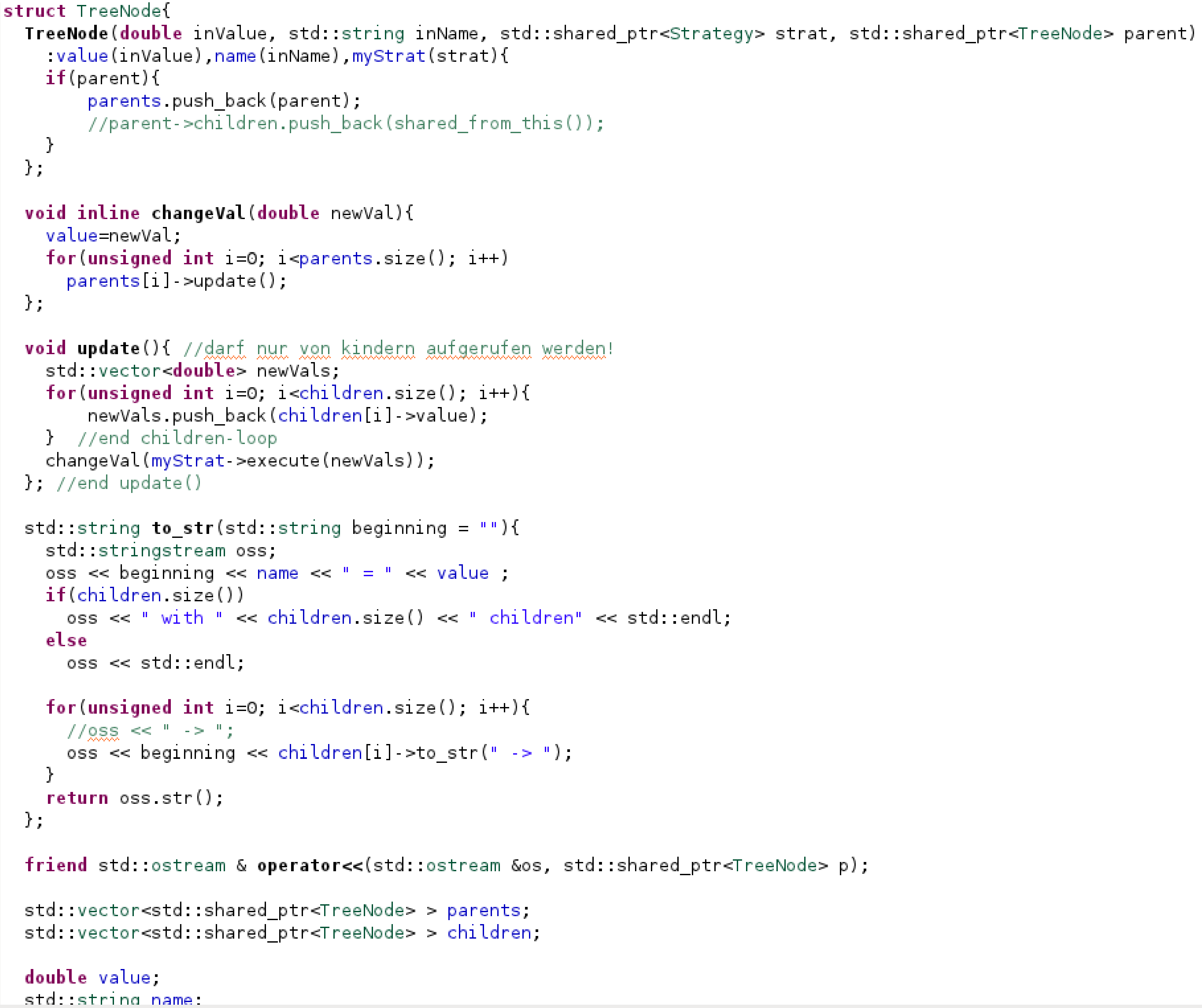
- Slaves need ip & port of master for communication and established tcp/ip connection

Geneva Individual Networked Mode

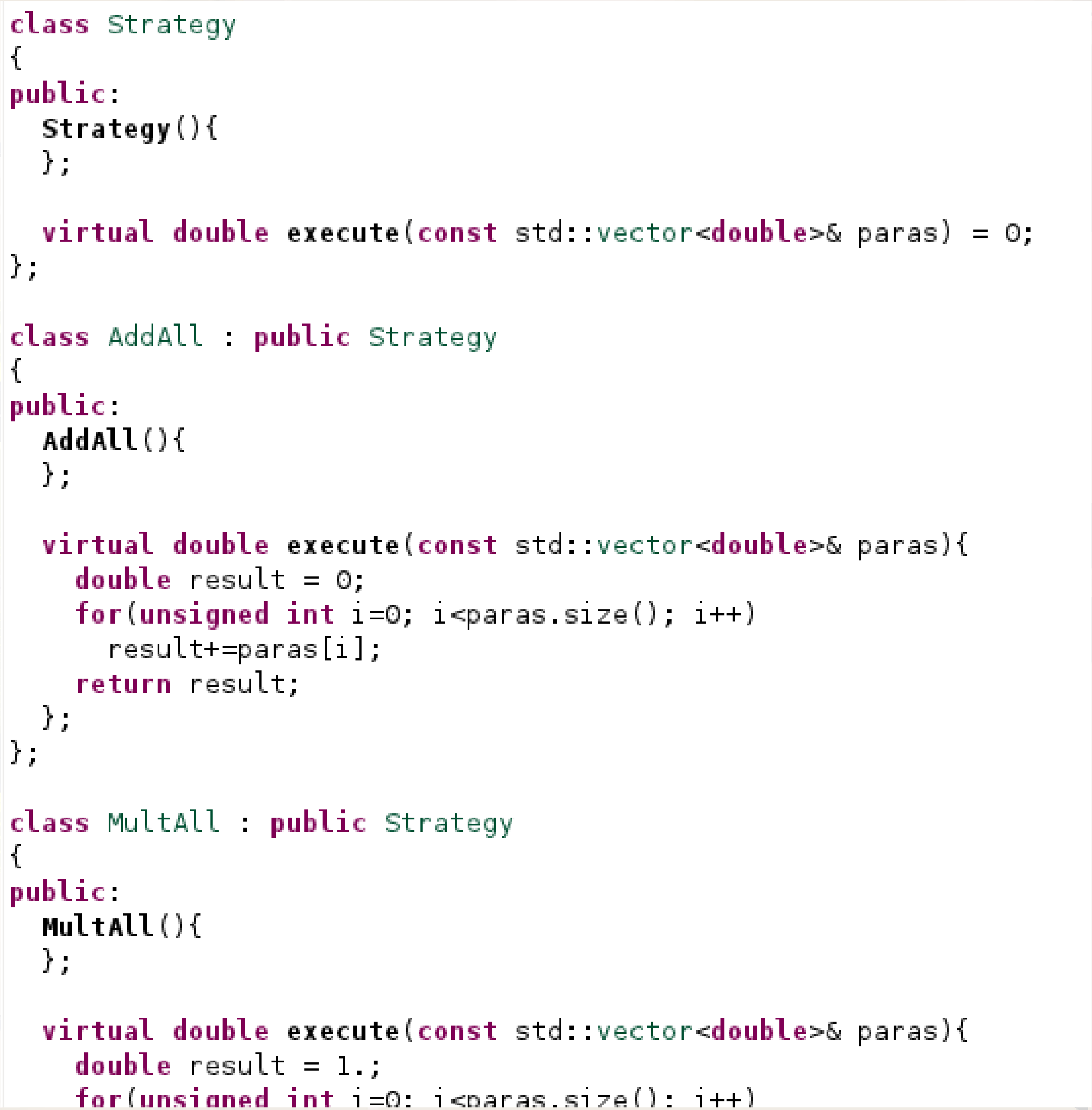


- Master sets up Individual **as usual**
- Master sends Individual to slaves (serialized)
- Slaves send best Individual back
- Repeat

=> Provide control Parameter (LH) constantly in each process, new Individuals get LH not from serialized information but from static object



```
struct TreeNode{
  TreeNode(double inValue, std::string inName, std::shared_ptr<Strategy> strat, std::shared_ptr<TreeNode> parent)
    :value(inValue),name(inName),myStrat(strat){
    if(parent){
        parents.push_back(parent);
        //parent->children.push_back(shared_from_this());
    }
  };

  void inline changeVal(double newVal){
    value=newVal;
    for(unsigned int i=0; i<parents.size(); i++)
      parents[i]->update();
  };

  void update(){ //darf nur von kindern aufgerufen werden!
    std::vector<double> newVals;
    for(unsigned int i=0; i<children.size(); i++){
        newVals.push_back(children[i]->value);
    }  //end children-loop
    changeVal(myStrat->execute(newVals));
  }; //end update()

  std::string to_str(std::string beginning = ""){
    std::stringstream oss;
    oss << beginning << name << " = " << value ;
    if(children.size())
      oss << " with " << children.size() << " children" << std::endl;
    else
      oss << std::endl;

    for(unsigned int i=0; i<children.size(); i++){
      //oss << " -> ";
      oss << beginning << children[i]->to_str(" -> ");
    }
    return oss.str();
  };

  friend std::ostream & operator<<(std::ostream &os, std::shared_ptr<TreeNode> p);

  std::vector<std::shared_ptr<TreeNode> > parents;
  std::vector<std::shared_ptr<TreeNode> > children;

  double value;
  std::string name;
```



```
class Strategy
{
public:
  Strategy(){
  };

  virtual double execute(const std::vector<double>& paras) = 0;
};

class AddAll : public Strategy
{
public:
  AddAll(){
  };

  virtual double execute(const std::vector<double>& paras){
    double result = 0;
    for(unsigned int i=0; i<paras.size(); i++)
      result+=paras[i];
    return result;
  };
};

class MultAll : public Strategy
{
public:
  MultAll(){
  };

  virtual double execute(const std::vector<double>& paras){
    double result = 1.;
    for(unsigned int i=0; i<paras.size(); i++)
```