Smoofit status & discussion

Analysis tools task force

Sébastien Wertz

December 1st, 2021

Motivation

- While HC is a complete & mature tool, I grew frustrated with:
  - Lack of API: datacards $\sim$ never written by hand, require scripts/additional tool (e.g. CombineHarvester) to generate
  - Lack of API: command-based $\rightarrow$ often long hard-to-read/-remember commands, have to write scripts which generate scripts...; doing anything not in the set of available "methods" difficult/requires expert input
  - Some newer features not user-friendly: unfolding (regularization); partial nuisance parameter correlations require external script
  - For complex fits, it can be slow
  - Minuit often runs into trouble, requires black magic to get fit to converge & obtain uncertainties
- Would be great to profit from automatic differentiation for fitting (+covariance matrix)
- Having a simple (python) API to build the models & run the fits and diagnostics could increase productivity

Formulation of the problem

- Parameters of interest: $\vec{\mu}$, nuisance parameters: $\vec{\alpha}$
- n bins and p processes
- Predicted yields in bin j for process i: $N_{ij}(\vec{\mu}, \vec{\alpha})$
- Observed yields in bin j: D_j
- Auxiliary observables $\vec{\mathbf{B}}$ (for frequentist treatment of nuisances)
- NLL written as:

$$L(\vec{\mathbf{D}}, \vec{\mathbf{B}}|\vec{\mu}, \vec{\alpha}) = -\sum_{j=1}^{n} \text{LogPoi}\left(D_j \middle| \sum_{i=1}^{p} N_{ij}(\vec{\mu}, \vec{\alpha})\right) + C(\vec{\mathbf{B}}|\vec{\alpha}) + R(\vec{\mu}, \vec{\alpha}, \vec{\mathbf{D}}, \vec{\mathbf{B}})$$

- Only assume we can write: $N_{ij}(\vec{\mu}, \vec{\alpha}) = \prod_k f_{kij}(\vec{\mu}, \vec{\alpha}) N_{ij}^0$
  - f_{kij} are arbitrary user-supplied functions
  - Systematics, unfolding, EFT fits, ...
- $C(\vec{\mathbf{B}}|\vec{\alpha})$ is nuisance parameter constraint, including arbitrary correlations (multi-dimensional Gaussian), in most cases $\vec{\mathbf{B}} = \vec{\mathbf{0}}$
- $R(\vec{\mu}, \vec{\alpha}, \vec{\mathbf{D}}, \vec{\mathbf{B}})$ is arbitrary user-supplied constraint, e.g. for unfolding regularization ($\rightarrow R = R(\vec{\mu})$)

Smoofit: building a model

Only four building blocks needed to build model:

- *`Variable`*: represent scalar or vector-valued parameters
  → POIs or nuisances
- *`Process`*: represent processes → can be "vectors" too, i.e. group multiple processes together → unfolding, EFT
- *`ChannelContrib`*: define how a process contributes to a channel
  → specify yields, dependency on parameters
- *`Model`*: takes all the *`ChannelContrib`*, defines full likelihood
- Note: objects are "stateless": no pre-/post-fit state

Abritrary yield dependency on parameters:

- Pass any callable, specify which *`Variable`* are used
- Only requirement: write function using JAX for autodiff + jitting
- Pre-defined functions for systematic interpolations
- Functions are $n \to m$: parameter arrays can match to "sub-processes", template bins, etc.

Smoofit quick showcase

- Building a simple model with one channel, one signal, one background:

```
from smoofit.model import Variable,Process,Model,ChannelContrib
import numpy as np; import jax.numpy as jnp

sig = Process("sig")
bkg = Process("bkg")

# implicitly declares channel called "SR"
sig_sr = ChannelContrib("SR", np.array([50., 1.]))
sig.add_contrib(sig_sr)
bkg_sr = ChannelContrib("SR", np.array([500., 500.]))
bkg.add_contrib(bkg_sr)

mu = Variable("mu", 1., lower_bound=0.)
sig.scale_by(mu) # just scale the signal yields linearly with mu

model = Model()
model.add_proc(sig)
model.add_proc(bkg)

# Collect all functions and parameters
model.prepare()
# obtain gradient, trigger compilation of NLL+gradient
model.compile()
```

Smoofit quick showcase

- Generate Asimov toy, run a fit:

```
# Obtain array with all declared parameters (here it's only mu), with some fixed
    values
values = model.values_from_dict({mu: 0.5})
# Predict yields = generate Asimov toy using those parameter values:
asimov_yields = model.pred(values) # --> [525.  500.5]

# Fit the toy
best_fit = model.fit(asimov_yields)
# best_fit.x --> [0.50001344]
# best_fit.hess_inv -> [[0.20991246]] <-- analytical covariance matrix
# => so fitted mu = 0.50 +- 0.46

# Profiled bounds ("Minos")
up,down,_,_ = model.minos_bounds(mu, best_fit, asimov_yields)
# => mu = 0.50 +0.46 -0.45
```

Systematic uncertainties

- Supported: (asymmetric) log-normal, shape uncertainties (vertical interpolation with continuous second derivatives, same as in HC)
- Automatic bin-by-bin stat. uncertainties (Barlow-Beeston lite)
- Systematics are registered with channel contributions:

```
# Declare nuisance parameter, default value = 0
lnN_lumi = Variable("lnN_lumi", 0., nuisance=True)
# 5% uncertainty for signal; similar interface for shape uncertainties
sig_sr.add_lnN(lnN_lumi, 1.05)
# variance of event weights for MC stat. uncertainties
sig_sr.register_sumw2(np.array([5., 0.1]))

# Re-run as previous slides (compile, fit):
# mu = 0.50 +- 0.60 (uncertainties from Hessian)
# mu = 0.50 +0.62 -0.50 (profiled uncertainties) -> lower bound is respected
```

- Support arbitrary correlations between nuisances:

```
lumi_2016 = Variable("lnN_lumi_2016", 0., nuisance=True)
lumi_2017 = Variable("lnN_lumi_2017", 0., nuisance=True)
# ...
model.correlate_nuisances(lumi_2016, lumi_2017, 0.5) # 50% correlated
# compile etc.
```

Array processes and variables

- Variables and processes can also be arrays ($\rightarrow$ “sub-processes”, “sub-variables”) $\rightarrow$ mostly useful for unfolding, EFT fits

```
mu = Variable("mu", [1., 1.], sub_names=["mu_1", "mu_2"])

# Stack two signals into one 2D array (rows=processes)
sigs = Process("sigs", [[50., 1.], [1., 25.]], sub_procs=["sig_1", "sig_2"])

# sig_1 will be scaled by mu_1 component, sig_2 by mu_2
sigs.scale_by(mu)

model.prepare(); idxs = mu.sub_idxs
# add (dumb) constraint on mu for regularization
#  (full TUnfold-style operator also available, supports non-uniform bin widths)
model.add_constraint(lambda x,_,_: -0.1 * jnp.sum((x[idxs] - 1.)**2))
```

- Array variables can scale across sub-processes or across template bins (useful e.g. for bin-by-bin ABCD method)
- Variables can scale whole process or only specific channels (useful for data-driven estimation)

```
bkg_sr.scale_by(Variable("bkg_scale", 1.)) # only scale in specific channel
sig.scale_by(Variable("sig_scale", 1.)) # scale whole process
```

EFT fits

- "Array" variable and process combined with arbitrary scaling function
  $\rightarrow$ easy to implement EFT morphing
- Provided out-of-the box: dim.-6 EFT with n operators
  $\rightarrow$ contributions (cross sections, yields...) assumed to scale like:

$$H(\vec{c}) = H_0 + \sum_{i=1}^{n} c_i H_i + \sum_{i=1,j=1,i\leq j\leq n} c_i c_j H_{ij} \tag{1}$$

- Where:
  - H_0 is the SM prediction
  - $\vec{c}$ is array of Wilson coefficients
  - H_i are the interferences between the SM and operator i
  - H_{ij} are pure-EFT contributions (operator-squared and operator-operator interfence)

EFT fits

- Input needed: set of templates $T_k = H(\vec{c}_k)$ with $1 \leq k \leq M$, $M = 1 + n + n(n+1)/2$
- E.g. T_k are separate histograms filled using event weights for each $\vec{c}_k$
- If more than M weights available (from event generation), can fit events first using (1) and choose M basis points when filling templates
- Can then morph T_k basis into the prediction for any value of $\vec{c}$:

$$H(\vec{c}) = \sum_{k=1}^{M} f_k(\vec{c}) T_k$$

where:

$$f_k(\vec{c}) = \sum_{l=1}^{M} w_j(\vec{c}) W_{jk}^{-1},$$

$$\vec{w}(\vec{c}) = (1,\, c_1, \ldots, c_n,\, c_1 c_1, c_1 c_2, \ldots, c_1 c_n, c_2 c_2, c_2 c_3, \ldots, c_n c_n),$$

$$W_{jk} = w_j(\vec{c}_k)$$

EFT fits in Smoofit

- Use in smoofit:

```
# Variable for Wilson coefficients
c_var = Variable("Wilson", [0., 0.], sub_names=["ctG", "cphiG"])

# Stack morphing input templates into one 2D array (rows=processes, here 6)
sigs = Process("sigs", np.array([basis1, basis2, ..., basis6]))

# Matrix with M rows = values of Wilson coefficients used to fill basis templates
# e.g. c1 = [1., 0.] was used for basis1 etc.
basis_values = np.array([c1, c2, ..., c6])

# Build and register morphing function
eft_scaling = Dim6EFTMorphing(c_var, basis_values)
sigs.scale_by_fn(eft_scaling, c_var)
```

- *eft_scaling* is a callable, returns $f_k(\vec{c})T_k$ yield array of shape *(M,#bins)*

```
# Example:
eft_scaling(np.array([1., 0.])) # -> returns [basis1, [0...], ..., [0...]]
```

- Sum of yields over $k = 1 \ldots M$ done with sum over all (sub-)processes

API (dis)advantages

Python API enables to easily:

- Compute predicted yields for given values of POIs/nuisances
- Generate toys using values of POIs/nuisance (or from different model)
- Fit specific toy, inspect results
- Sample from covariance matrix, batch-evaluate predicted yields
  $\rightarrow$ easy post-fit uncertainties
- ...
- Note: all of that possible in Combine...but not convenient

Major difficulty:

- Abritrary functional dependencies $\rightarrow$ how to serialize or combine models?

Minimization

- Gradient descent methods (as with DNNs) converge slowly
  - Much smaller number of parameters than DNNs
    $\rightarrow$ can use more computationally intensive quasi-Newton methods
  - Also, ability to incorporate (non-)linear constraints (for profiling) is a +
- Currently using NumPy's `trust-constr` method for minimization with arbitrary constraints
  - Trust-region method with SR1 approximation of the Hessian
  - Observed to work well, only a few isolated instances of fit not converging with *default* parameters
  - Con: significant overhead in pure NumPy, not run on GPU
- Use pure-JAX minimizers?
  - Expect gain when running on GPU
  - Vectorize fits over toys?
  - Differentiate "through" minimizer?
  - For combineTF, work by Josh B. on implementing minimizers in pure TensorFlow $\rightarrow$ porting to JAX could be useful?
  - BFGS now available in JAX, but no support for constraints or bounds; perhaps try jaxopt ?

Model ”preparation“ + performance

- JAX (jitting) doesn’t like loops...
- When ”preparing“ the model, regroup processes, channels and systematics → Obtain several (*) large 3D arrays:
  `nSyst x nProcesses x [nBins across channels]`
- Automatically pad arrays as appropriate when:
  - (Sub-)process doesn’t contribute to channel
  - Systematic doesn’t affect (sub-)process/channel
  - User-supplied function affects specific (sub-)process/channl
- Pre-processing using plain numpy (non-jitted JAX too slow)
- Comes at expense of higher memory usage
- (*) several arrays: yields, sumw2; for shape systematics: interpolate integral & normalized shape separately

Conclusions

- Basic functionalities for fits and diagnostics in place, reasonable performance
  - Being used for unfolding $\mathrm{t\bar{t}b\bar{b}}$ (AN-21-040) – note: no combination needed (or possible) there
  - Validated against Combine
- TODO list (no showstopper):
  - CLs limits, GoF tests, Feldman-Cousins intervals
  - Performance improvements (memory usage), refactoring ”inference“ API
- Obviously not meant as replacement for Combine; much more modest project than pyhf
  - Implement missing features as needed/asked, but no plan for wide support
  - Could still be useful for people in ”edge“ cases
  - Hope some ideas/features will find their way elsewhere
- Documentation, repository

Backup

Which backend?

PyTorch

- Eager execution + gradient tape: user-friendly but "slow"
- Mechanism for compiling model (TorchScript), but not user-friendly
- Second derivatives (for Hessian) are a bit tricky

TensorFlow 2

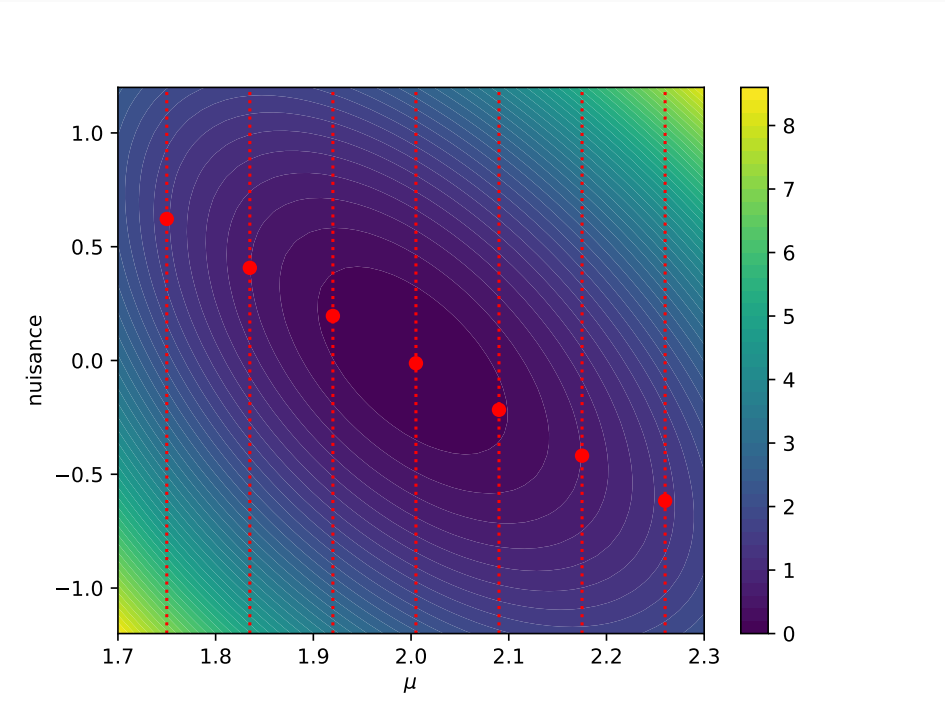
- Eager execution + gradient tape: user-friendly but "slow"
- Easy mechanism for creating static graph (`tf.function`) $\rightarrow$ faster
- Large dependency (>1 GB)

JAX

- "NumPy on steroids": NumPy "replacement" with support for GPU, auto-differentiation, auto-vectorization
- Mechanism for JITting functions (XLA) $\rightarrow$ super fast (fuses operations)
- Much lighter dependency (<200 MB)

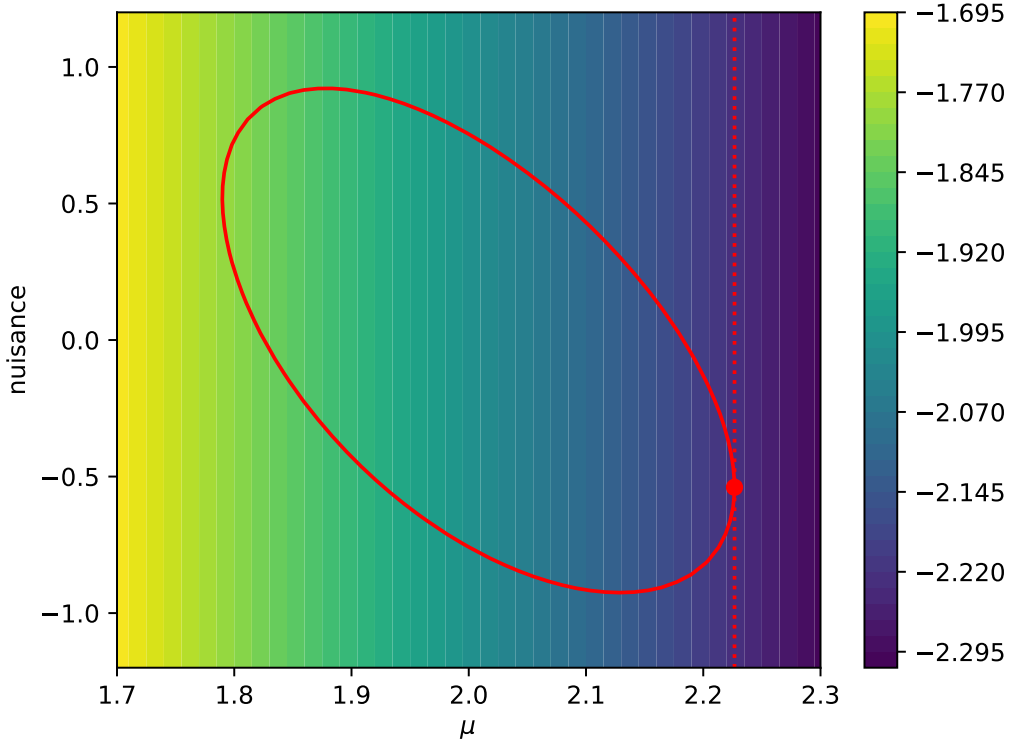
Implementation of profiling

- Profiling: minimize NLL while fixing one parameter
- Avoid re-defining function to be minimized (avoid re-tracing + jitting of gradient) $\rightarrow$ set gradient entry to zero while minimizing, or:
- Use constrained minimization with "full" NLL
- Profiled NLL at $\mu_0 \rightarrow$ minimize NLL while constraining $\mu = \mu_0$ (linear) $\rightarrow$ gradient of objective: autodiff; of constraint: analytical
- Used for profiled NLL scans, nuisance impacts



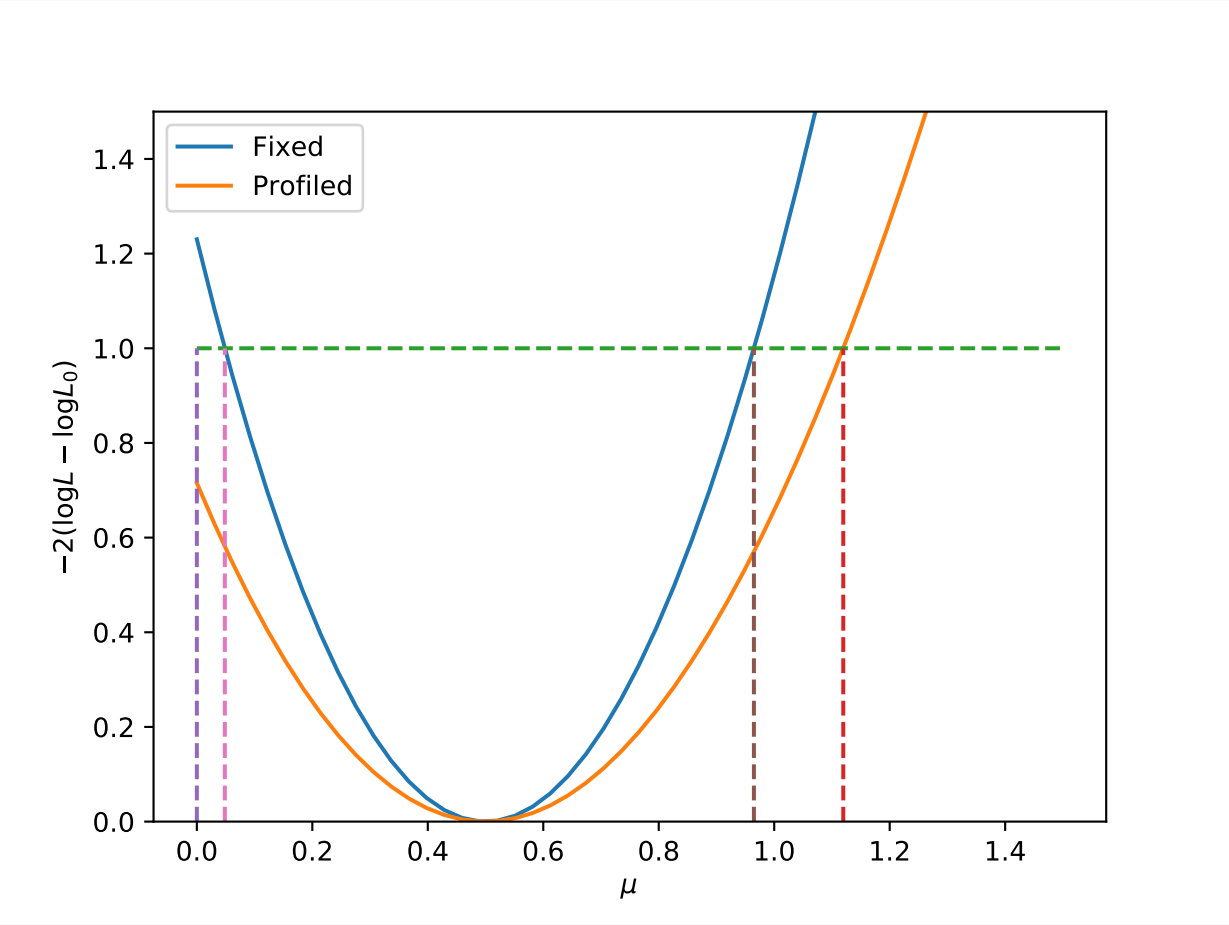
“Minos” uncertainties

- Confidence intervals obtained from profiled Δ NLL crossings
- Upper bound on μ can be obtained by minimizing $f(\mu, \alpha) = -\mu$ under nonlinear constraint $L(\mu, \alpha) \leq L_0 + 1/2$ (where L_0 = best-fit NLL)
  $\rightarrow$ gradient of objective: analytical; of constraint: autodiff, reuse ∇(NLL)
- Can work in any direction (e.g. for 2D contour, parametrize by polar angle)
- Also used for nuisance pulls



NLL scans

- Profiled and stat.-only (nuisances frozen) scans of NLL vs. μ
- 1-sigma uncertainties (vertical lines) are from direct determination, not from “explicit” NLL scan



Other features

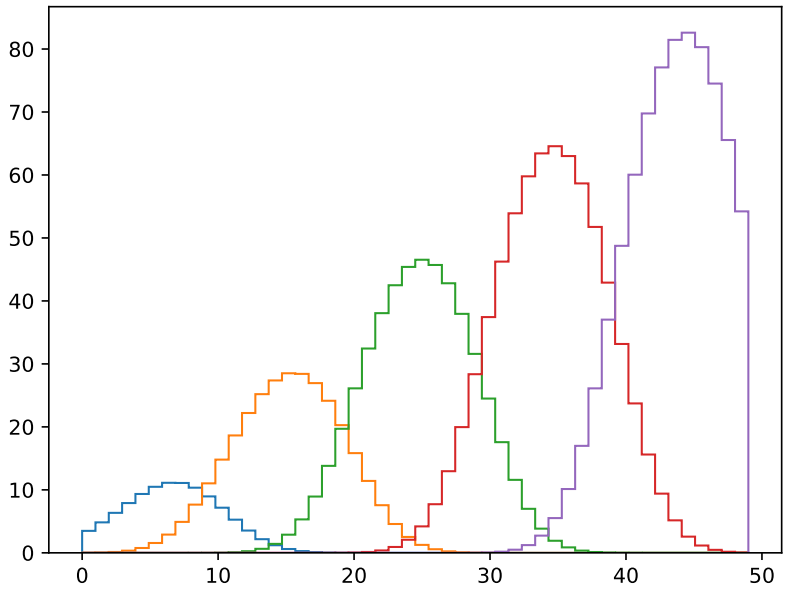
- Pulls: Minos bounds on specific nuisance
- Impacts: shift in μ when nuisance shifted by $\pm 1\sigma$ (=profiling)
- Batch-generated frequentist toys
- Fast sampling from covariance matrix
  $\rightarrow$ fast (batch) evaluation of ***Process.pred*** and ***Model.pred***
  $\rightarrow$ postfit shapes and uncertainties, separately for each process or for whole model (including effect of BB-lite parameters)

```
# Pulls
pull_u,pull_d,_,_ = model.minos_bounds(lnN_lumi, best_fit, obs)

# Impacts
impact_up = model.profile({lnN_lumi: pull_u}, best_fit, obs).x[0] - best_mu
impact_down = model.profile({lnN_lumi: pull_d}, best_fit, obs).x[0] - best_mu

# Throw 10000 frequentist toys at once
nObs, nGlob = model.toy_pred(values, 1e4), model.toy_glob(values, 1e4)

# Postfit shapes from 10000 covariance matrix samples
batch_values = model.sample_from_covariance(best_fit, 1e4)
batch_pred = model.batch_pred(batch_values)
postfit_up = np.quantile(batch_pred, 0.68, axis=0)
postfit_down = np.quantile(batch_pred, 0.32, axis=0)
```

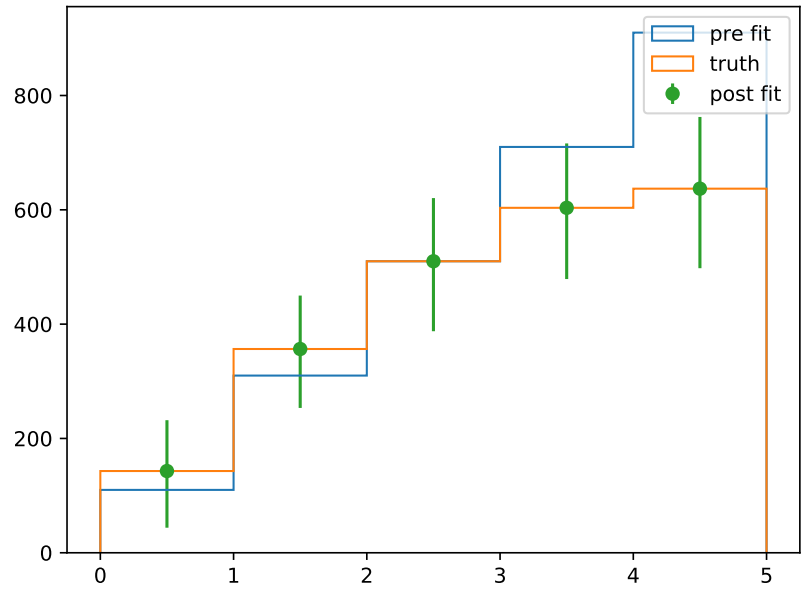
Unfolding example

- Example with 50 reco-level bins, 5 gen-level bins (POIs), 5 backgrounds, 5 nuisance parameters
- $\sim$ 1 minute for fit, covariance matrix, Minos intervals on each POI, pulls and impacts on each POI

Reco-level distributions for each gen-level bin (pre-fit):

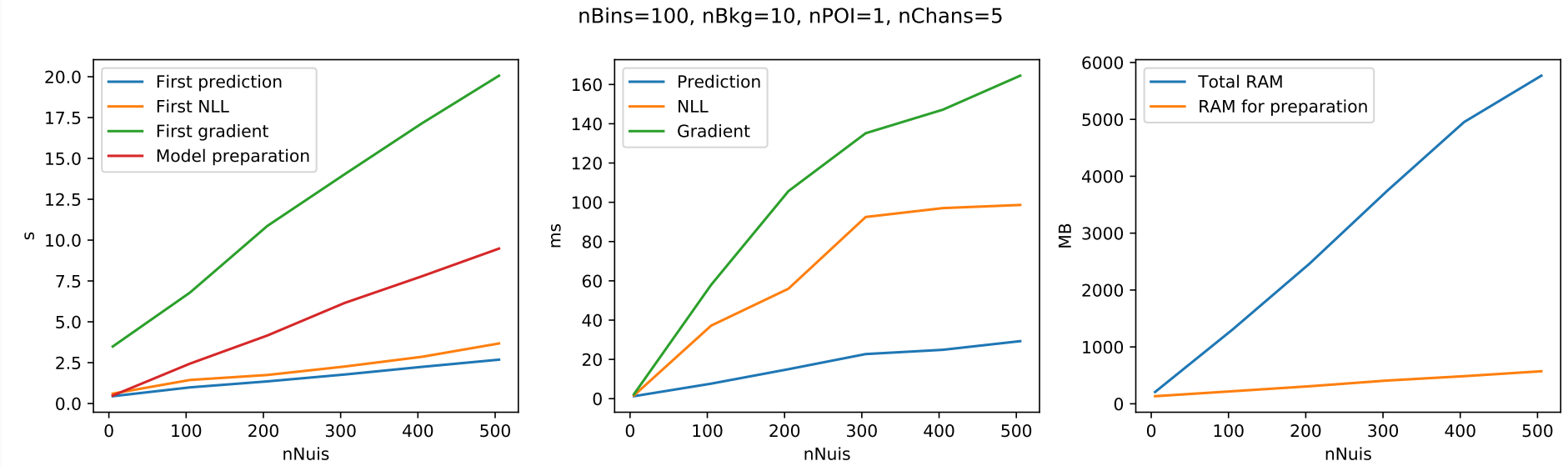


Gen-level results, different shape injected ("truth"):

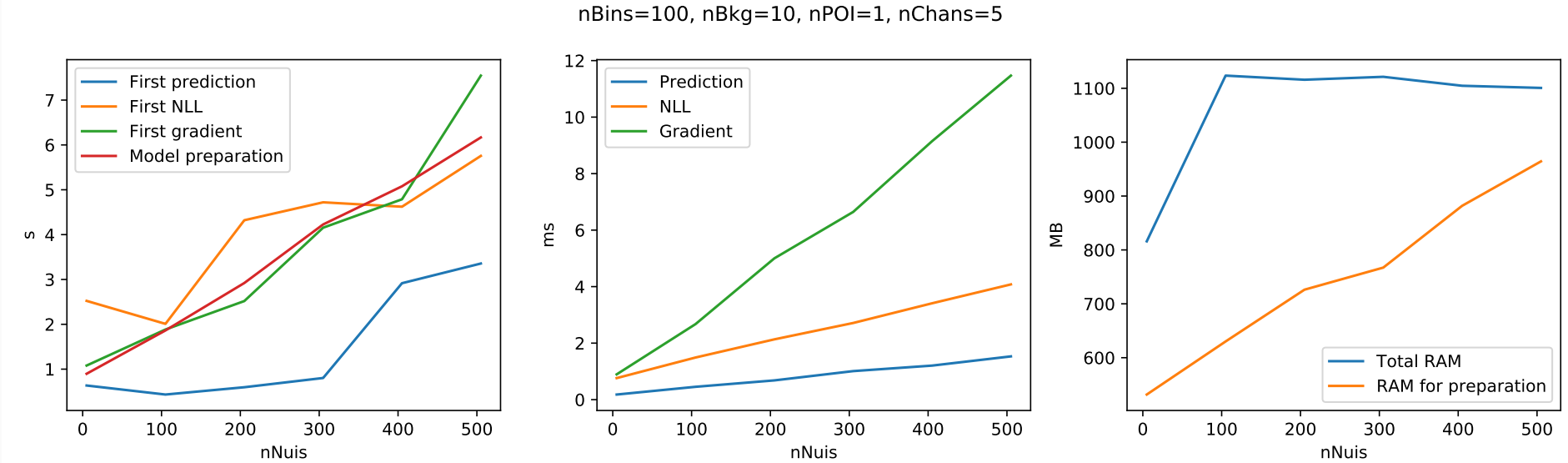


Performance scaling: number of nuisances

CPU (2xE5-2695 – 24 physical cores):

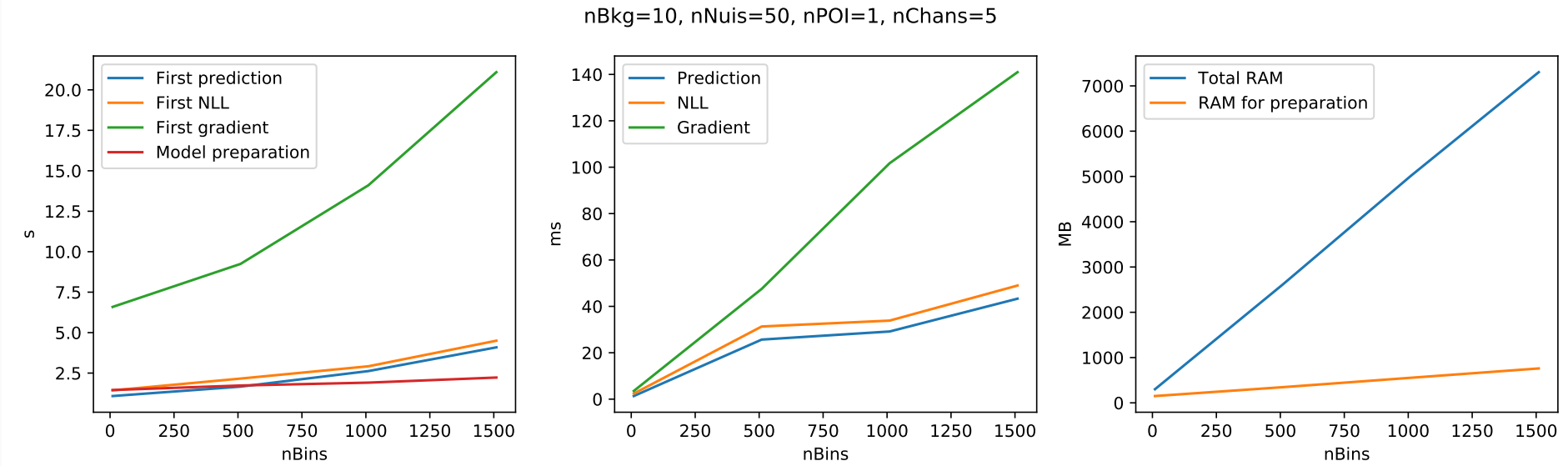


K40 GPU (RAM usage on host only):

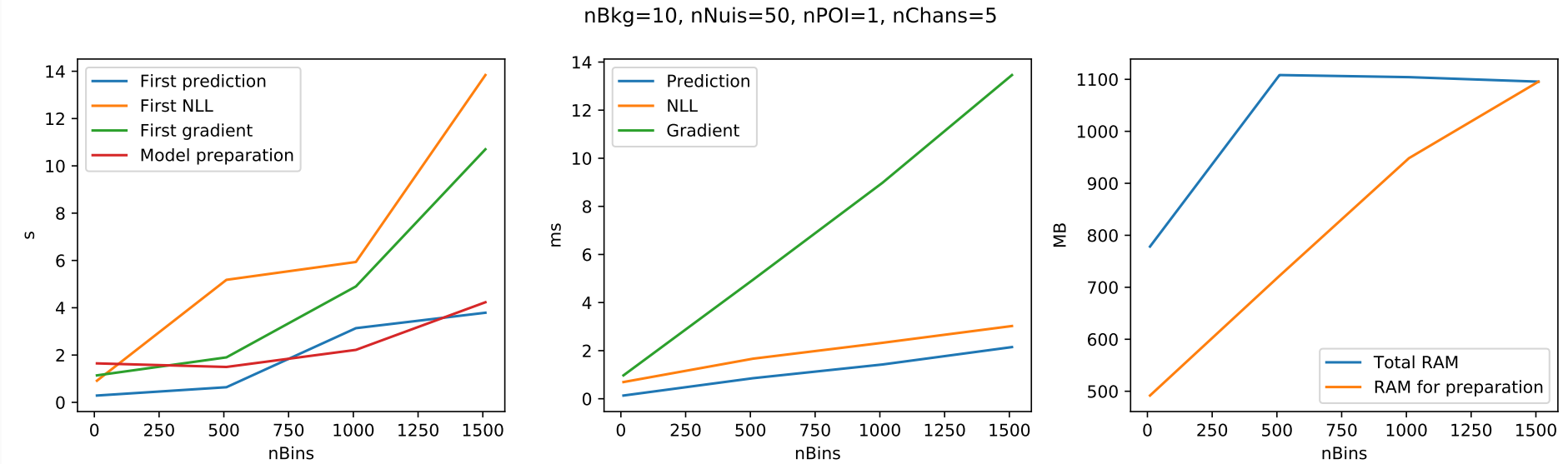


Performance scaling: number of bins

CPU (2xE5-2695 – 24 physical cores):



K40 GPU (RAM usage on host only):



Performance: practical case

For $\mathrm{t\bar{t}b\bar{b}}$ differential (AN-21-040):

- 6 POIs, 255 nuisances, 192 bins
- Fit toy (*) + Hessian + profiled bounds on 6 POIs: 4 min
- Nuisance pulls (62 nuisances (**)): 30 min
- Nuisance impacts (6 POI x 255 nuisances): 2 hours
- (*): for Asimov toy: fit + Hessian = 20s; profiled uncertainties = 80s
- (**): pulls for 192 BB-lite nuisances taken from Hessian, others from profile NLL scan
- Using ~ 16 CPUs

Asymmetric lnN interpolation

- N_i are yields in bin i, α is nuisance parameter
- Symmetric log-normal uncertainty K: $N_i \to K^{\alpha} \cdot N_i$
- Asymmetric log-normal K_{up}, K_{down}:

$$
\begin{cases}
K_{\text{up}}^{\alpha} \cdot N_i & \text{if } \alpha > 1 \\
\left(\frac{\alpha}{4}(3 - \alpha^2)(K_{\text{up}} - K_{\text{down}}) + \frac{1}{2}(K_{\text{up}} + K_{\text{down}}) \right)^{\alpha} \cdot N_i & \text{if } |\alpha| \leq 1 \\
K_{\text{down}}^{\alpha} \cdot N_i & \text{if } \alpha < -1
\end{cases}
$$

- Gaussian constraint (with $\sigma = 1$, $\mu = B$) on α
- Continuous second derivative
- Exactly as in Combine (AFAIK)

Shape systematics interpolation

- N_i^{nom}: nominal yields in bin i; α: nuisance parameter
- Shape uncertainties: N_i^{up}, $N_i^{\text{down}} \rightarrow$ integrals I^{nom}, I^{up}, I^{down}, normalized shapes $\hat{N}_i^{\text{nom}}$, $\hat{N}_i^{\text{up}}$, $\hat{N}_i^{\text{down}}$ (e.g. $N_i^{\text{nom}} = I^{\text{nom}} \hat{N}_i^{\text{nom}}$)
- Interpolation of normalization using asymmetric lnN with $K^{\text{up}} = I^{\text{up}}/I^{\text{nom}}$, $K^{\text{down}} = I^{\text{nom}}/I^{\text{down}} \rightarrow$ function $F(\alpha)$, with $F(0) = I^{\text{nom}}$
- Interpolation of normalized shapes: $\hat{S}_i(\alpha) =$

$$\begin{cases} \hat{N}_i^{\text{up}} + (\alpha - 1)\left(\hat{N}_i^{\text{up}} - \hat{N}_i^{\text{nom}}\right) & \text{if } \alpha > 1 \\ \hat{N}_i^{\text{nom}} + \frac{\alpha}{2}\left(\hat{N}_i^{\text{up}} - \hat{N}_i^{\text{down}}\right) + \frac{1}{16}(3\alpha^6 - 10\alpha^4 + 15\alpha^2)\left(\hat{N}_i^{\text{up}} + \hat{N}_i^{\text{down}} - 2\hat{N}_i^{\text{nom}}\right) & \text{if } |\alpha| \leq 1 \\ \hat{N}_i^{\text{down}} - (\alpha + 1)\left(\hat{N}_i^{\text{down}} - \hat{N}_i^{\text{nom}}\right) & \text{if } \alpha < -1 \end{cases}$$

- Interpolated result: $N_i(\alpha) = F(\alpha)\hat{S}_i(\alpha)$
- Gaussian constraint (with $\sigma = 1$, $\mu = B$) on α
- Continuous second derivative
- Exactly as in Combine (AFAIK)