Making Stata estimation commands faster through automatic differentiation and integration with Python

Paul C Lambert[1,2]

[1]Department of Health Sciences, University of Leicester, UK
[2]Department of Medical Epidemiology and Biostatistics, Karolinska Institutet, Stockholm, Sweden

2021 Stata Conference
6^{th} August 2021

Slides: pclambert.net/pdf/Stata2021_Paul.Lambert.pdf

"My computer is too slow"

- We now have access to very large datasets with many rows.
- We may have tens of millions of individuals with multiple rows per individual.
- Fitting statistical models to large datasets can be frustrating slow.
- Applied statisticians / epidemiologists reluctant to leave they trusted software (R, SAS, Stata).

Aim

- This work aims to bring speed improvements when fitting statistical models in Stata
  - Almost invisible to the analyst.
  - Simple to implement for those developing the Stata commands.

Aim

- This work aims to bring speed improvements when fitting statistical models in Stata
  - Almost invisible to the analyst.
  - Simple to implement for those developing the Stata commands.
- The speed improvements are mainly due to using the Jax module within Python to,
  1. Make use of automatic differentiation.
  2. Provide fast compiled functions able to use multiple CPUs.

Where this is going?

- Use new optimizer, `mlad` rather than `ml`.
- Using Python to do some of the heavy computational work when fitting models - using multiple CPUs.
- Using automatic differentiation to avoid having to do the maths and programming to get Gradient and Hessian functions.

1,000,000 observations

```
timer on 1
strcs x1-x10, df(5) tvc(x1 x2 x3 x4 x5) dftvc(3)
timer off 1

timer on 2
strcs x1-x10, df(5) tvc(x1 x2 x3 x4 x5) dftvc(3) python
timer off 2
```

Where this is going?

- Use new optimizer, `mlad` rather than `ml`.
- Using Python to do some of the heavy computational work when fitting models - using multiple CPUs.
- Using automatic differentiation to avoid having to do the maths and programming to get Gradient and Hessian functions.

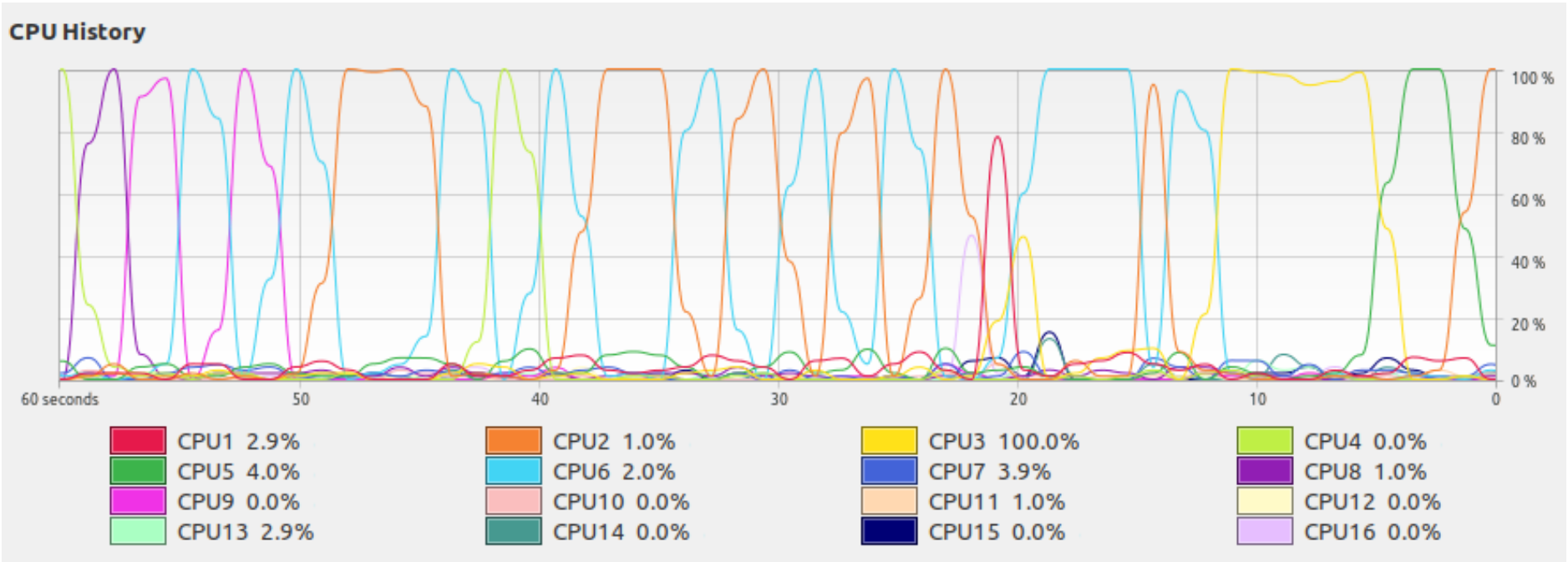
1,000,000 observations

```
timer on 1
strcs x1-x10, df(5) tvc(x1 x2 x3 x4 x5) dftvc(3)
timer off 1

timer on 2
strcs x1-x10, df(5) tvc(x1 x2 x3 x4 x5) dftvc(3) python
timer off 2


timer list
1:   2822.69 /        1 =    2822.6860
2:    110.56 /        1 =     110.5610

di 1 - 110.56/2822.69
0.96083169
```

Making more use of multiple CPUs

Stata BE or Stata SE

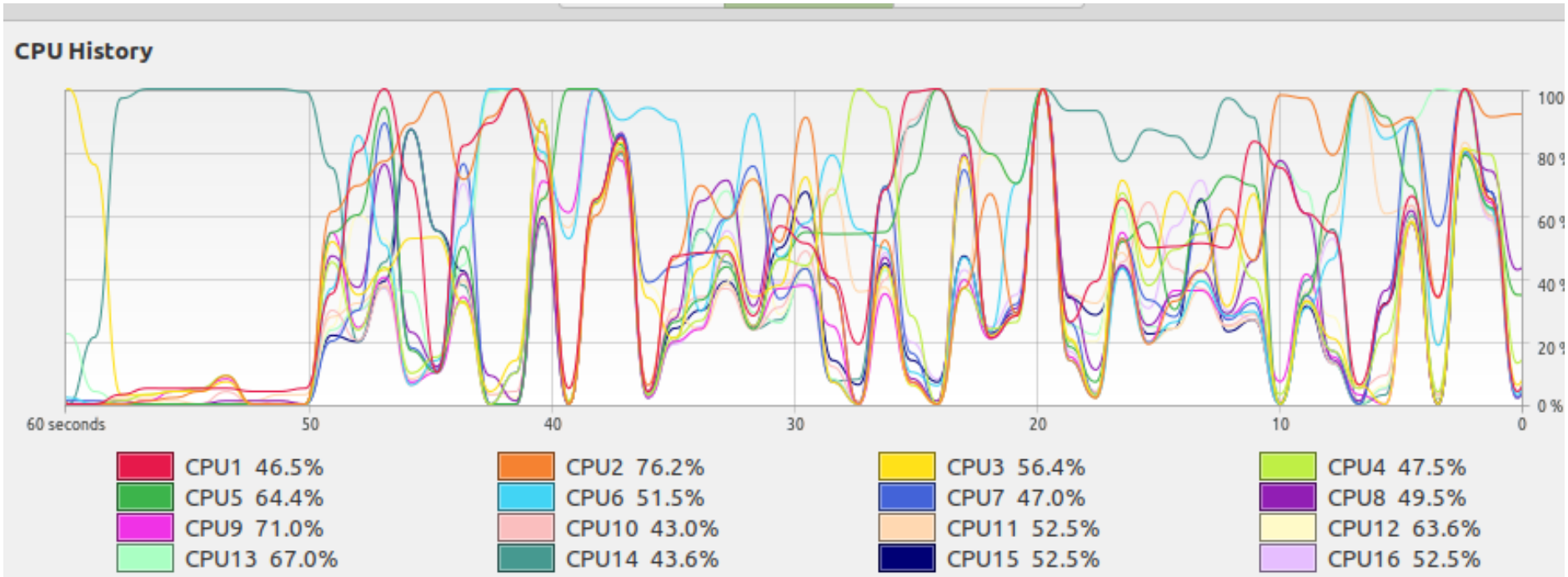
```
strcs x1-x10,  df(5) tvc(x1 x2 x3 x4 x5) dftvc(3)
```



Making more use of multiple CPUs

Stata BE or Stata SE

```
strcs x1-x10,  df(5) tvc(x1 x2 x3 x4 x5) dftvc(3) python
```



Maximum-likelihood estimation

- Stata has an excellent optimizer (`ml`) to estimate model parameters.
- Used in most of Stata's in-built model estimation commands.
- Used in most of user written model estimation commands.
- However, for complex models and large datasets estimation can be slow.
- Can invest in faster computer and/or Stata MP.

Estimation is an iterative process

Maximize the log-likelihood

$$\widehat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}}{\operatorname{argmax}}\ \ell_n(\boldsymbol{\beta}|Y, X)$$

Newton-Raphson

$$\widehat{\boldsymbol{\beta}}_{k+1} = \widehat{\boldsymbol{\beta}}_k + H\left(\widehat{\boldsymbol{\beta}}_k\right)^{-1} S\left(\widehat{\boldsymbol{\beta}}_k\right)$$

$\widehat{\boldsymbol{\beta}}_k$ - $1 \times p$ vector of parameters

$S\left(\widehat{\boldsymbol{\beta}}_k\right)$ - $1 \times p$ Score (Gradient) vector

$H\left(\widehat{\boldsymbol{\beta}}_k\right)$ - $p \times p$ Hessian matrix

Automatic Differentiation

$$S\left(\widehat{\beta}_k\right) = \frac{\partial \ell_n(\beta|Y,X)}{\partial \beta} \qquad H\left(\widehat{\beta}_k\right) = \frac{\partial^2 \ell_n(\beta|Y,X)}{\partial^2 \beta}$$

- Obtaining $S\left(\widehat{\beta}_k\right)$ and $H\left(\widehat{\beta}_k\right)$ can be computationally intensive.
- Automatic differentiation (AD) transforms code for one function into code for the derivative of the function
- Much, much faster than numerical integration (`d0`, `lf0`, `gf0`).
- In simple terms, AD is fast and you don't have to do the maths.
- Performed at compilation stage to give fast complied machine code.
- No AD procedure in Stata, but several in Python.

Jax (jax.readthedocs.io/en/latest/)

- Jax is a google "research project" for Python.
- It does lots, the main things of interest here
  - Automatic differentiation
  - Automatic vectorization of functions
  - Fast "Just-in-time" compilation
  - The fast, compiled code (XLA) will run on multiple CPUs (GPUs and TPUs).
- Active development with frequent updates / bug fixes.

Jax (jax.readthedocs.io/en/latest/)

- Jax is a google "research project" for Python.
- It does lots, the main things of interest here
  - Automatic differentiation
  - Automatic vectorization of functions
  - Fast "Just-in-time" compilation
  - The fast, compiled code (XLA) will run on multiple CPUs (GPUs and TPUs).
- Active development with frequent updates / bug fixes.

Good support for Linux and macOS (easy to install)
Less Windows support (need to build from source)

A new optimizer

- I have written an optimizer for Stata, `mlad`.
- Rather than a Stata program to define the likelihood the user needs to write a Python function.
- Automatic differentiation is used so the gradient and Hessian functions are calculated automatically using Jax.
- Likelihood, gradient and Hessian functions are compiled so fast and can make use of multiple processors.
- Makes use of Stata's `ml` command for setup, updating parameters and assessing convergence.
- All results are returned in Stata in standard `ml` format, so standard post-estimation tools are available.

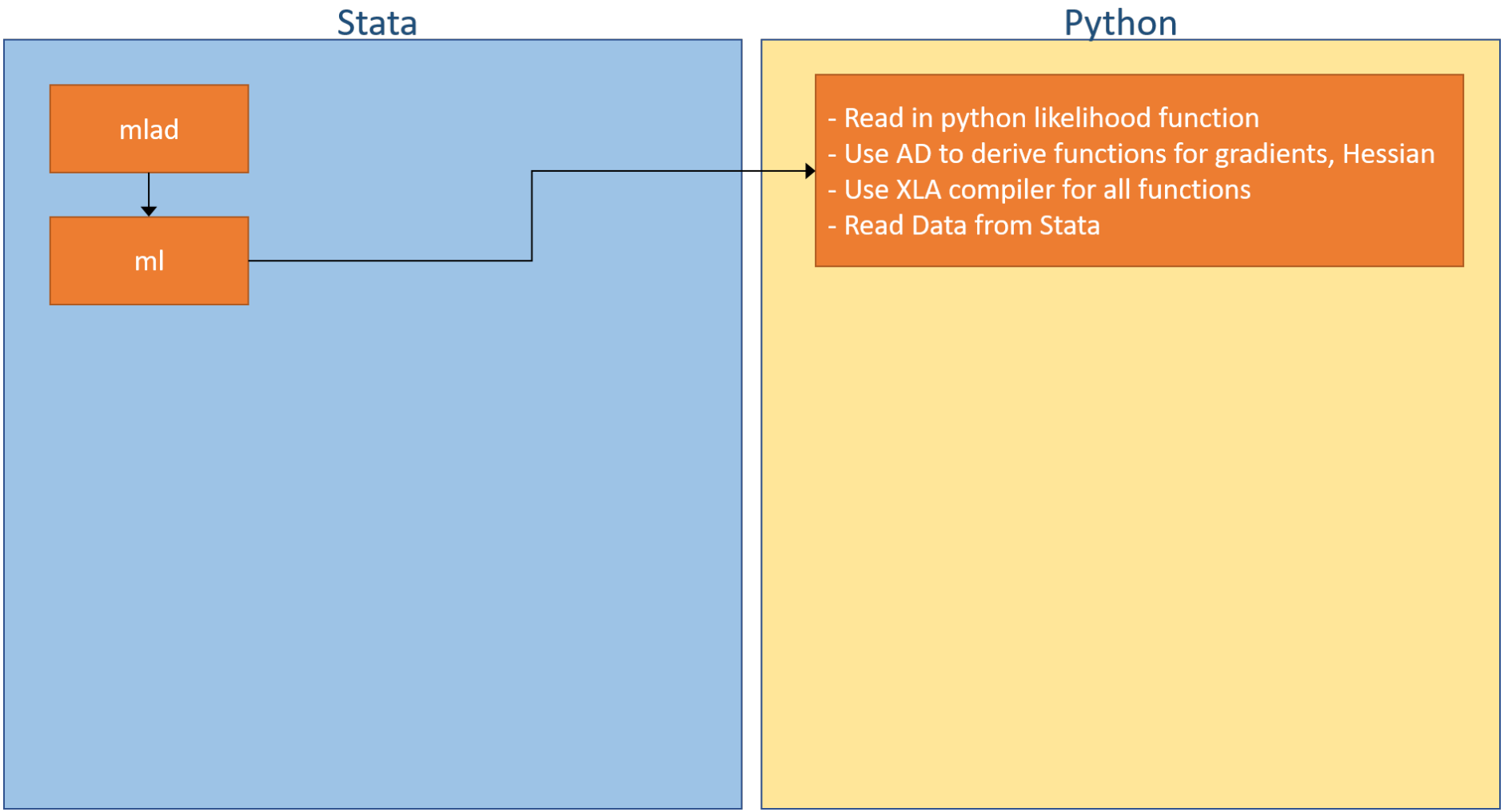
A new optimizer

- I have written an optimizer for Stata, `mlad`.
- Rather than a Stata program to define the likelihood the user needs to write a Python function.
- Automatic differentiation is used so the gradient and Hessian functions are calculated automatically using Jax.
- Likelihood, gradient and Hessian functions are compiled so fast and can make use of multiple processors.
- Makes use of Stata's `ml` command for setup, updating parameters and assessing convergence.
- All results are returned in Stata in standard `ml` format, so standard post-estimation tools are available.

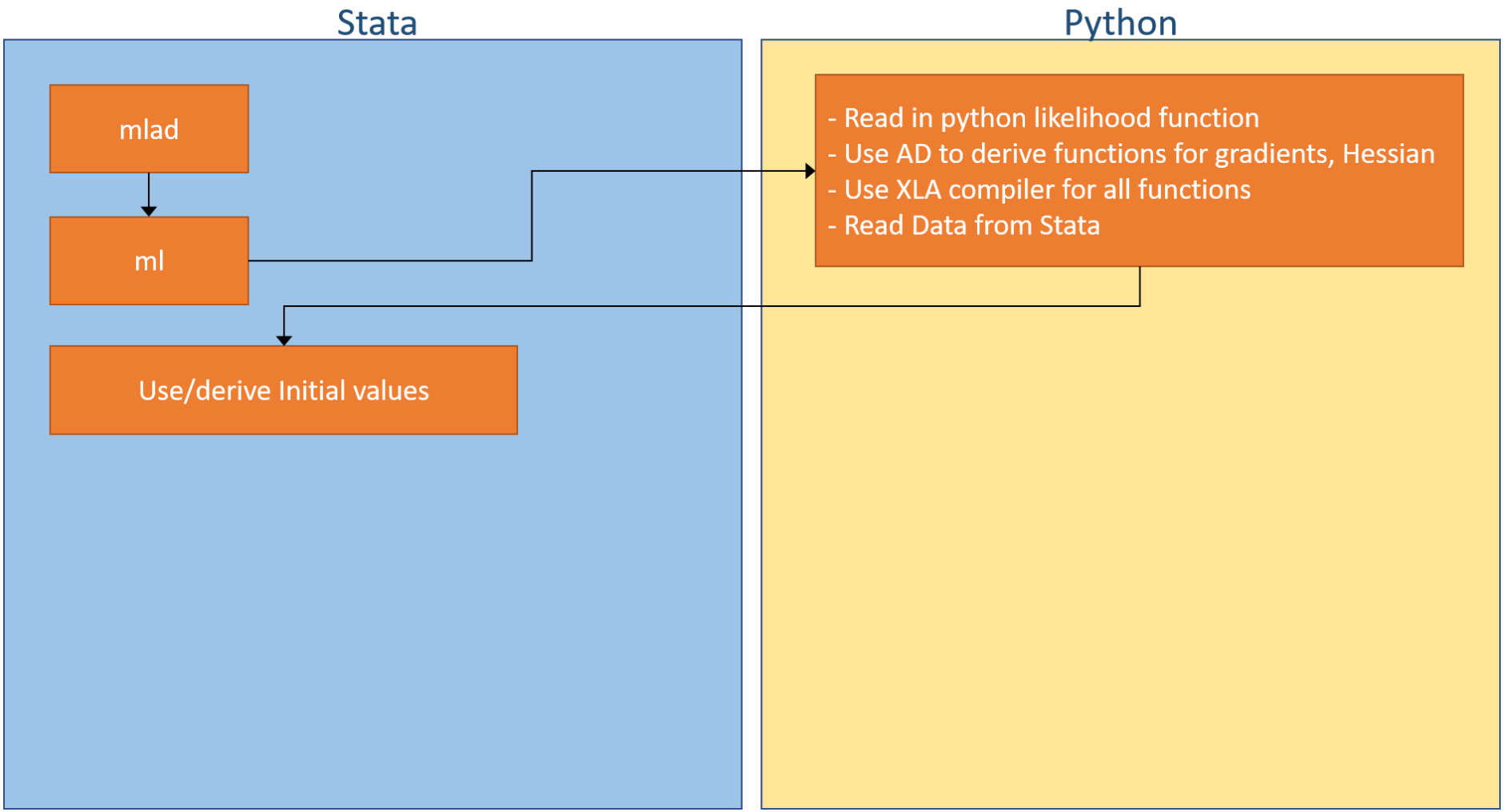
d0 $\rightarrow$ d2

We are writing a `d0` evaluator (in Python) and getting a `d2` evaluator for free.

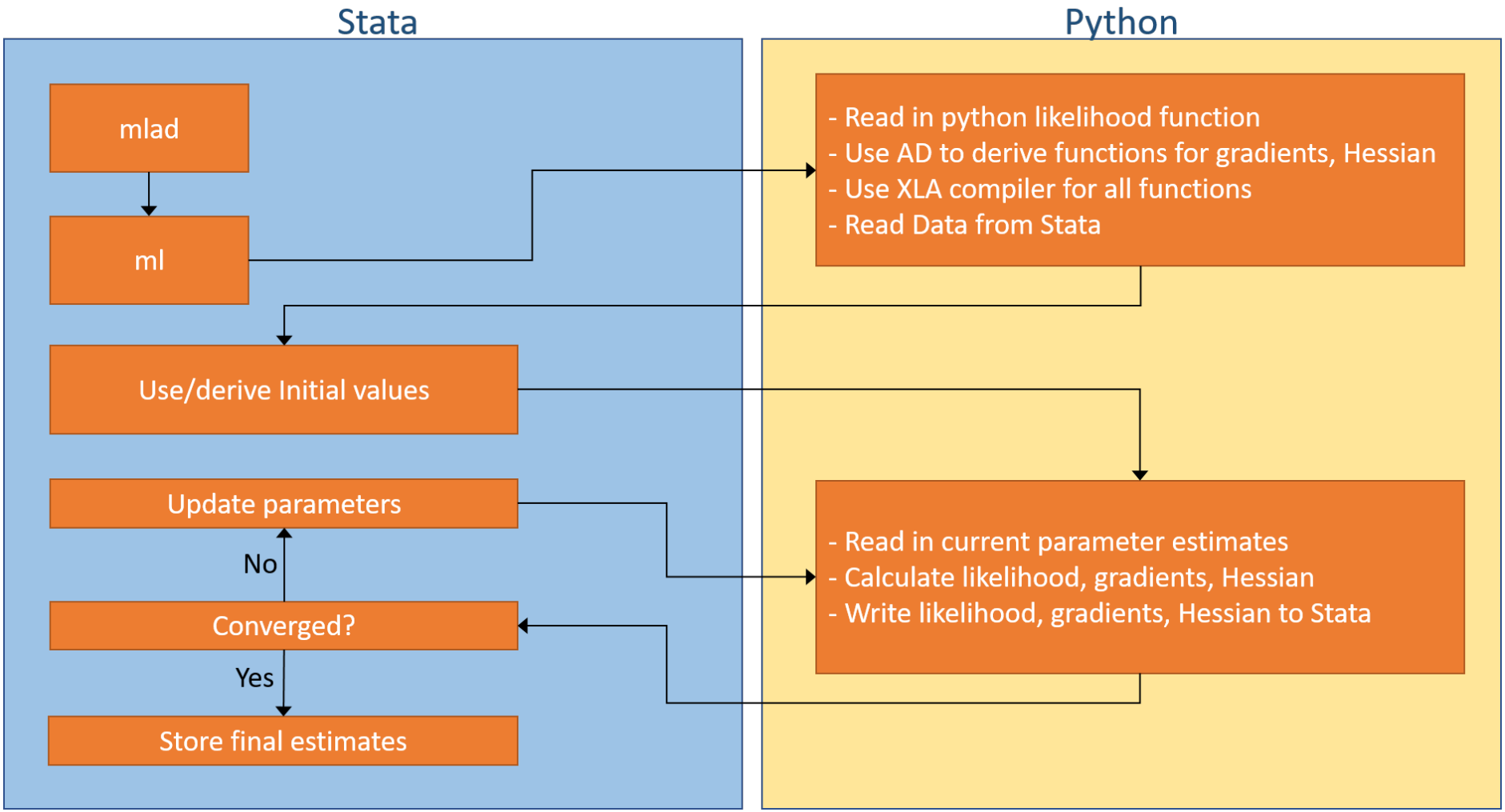
What is `mlad` doing?



What is mlad doing?



What is mlad doing?



My Computer

- Speed will vary between computers.
- All code in this talk is run on the following

- AMD Ryzen 7 3700X - 8 Cores (2 threads per core)
- CPU speed 4200 MHz
- Ram 32Gb
- Running Linux Mint 20.1 Cinnamon
- Cost $\approx$ £650

Example: A Weibull model

$$S(t) = \exp(-\lambda t^{\gamma}) \quad h(t) = \lambda\gamma t^{\gamma-1}$$

$$\ln \ell = \sum_{i=1}^{N} d_i \ln\left[h(t_i)\right] + \ln\left[S(t_i)\right]$$

Both λ and γ can depend on covariates.

$$\ln(\lambda) = X_1\beta$$

$$\ln(\gamma) = X_2\alpha$$

A Weibull model in Stata using ml (d0)

Stata d0

```
program weib_ll_d0
  version 16.1
  args todo b lnf g H

  tempvar lnlam lngam
  mleval `lnlam' = `b', eq(1)
  mleval `lngam' = `b', eq(2)

  mlsum `lnf' = _d*(`lnlam' + `lngam' + (exp(`lngam') - 1)*ln(_t)) - ///
                exp(`lnlam')*_t^(exp(`lngam'))
  if (`todo'==0 | `lnf'>=.) exit
end
```

A Weibull model in Stata using ml (d0)

Stata d0

```
program weib_ll_d0
  version 16.1
  args todo b lnf g H

  tempvar lnlam lngam
  mleval `lnlam' = `b', eq(1)
  mleval `lngam' = `b', eq(2)

  mlsum `lnf' = _d*(`lnlam' + `lngam' + (exp(`lngam') - 1)*ln(_t)) - ///
                exp(`lnlam')*_t^(exp(`lngam'))
  if (`todo'==0 | `lnf'>=.) exit
end
```

Using ml

```
ml model d0 weib_ll_d0 (lnlam: x1 x2 x3 x4 x5 x6 x7 x8 x9 x10) ///
                       (lngam: x1 x2 x3 x4 x5 x6 x7 x8 x9 x10) ///
                       , maximize
```

A Weibull model using mlad

Python file: weib_ll.py

```
import jax.numpy as jnp
import mladutil as mu

def python_ll(beta,X,wt,M):
  lnlam =  mu.linpred(beta,X,1)
  lngam  = mu.linpred(beta,X,2)
  gam = jnp.exp(lngam)

  return(jnp.sum(M["d"]*(lnlam + lngam + (gam - 1)*jnp.log(M["t"])) -
         jnp.exp(lnlam)*M["t"]**(gam)))
```

A Weibull model using mlad

Python file: weib_ll.py

```
import jax.numpy as jnp
import mladutil as mu

def python_ll(beta,X,wt,M):
  lnlam =  mu.linpred(beta,X,1)
  lngam  = mu.linpred(beta,X,2)
  gam = jnp.exp(lngam)

  return(jnp.sum(M["d"]*(lnlam + lngam + (gam - 1)*jnp.log(M["t"])) -
         jnp.exp(lnlam)*M["t"]**(gam)))
```

Using mlad

```
mlad  (lnlam: x1 x2 x3 x4 x5 x6 x7 x8 x9 x10)  ///
      (lngam: x1 x2 x3 x4 x5 x6 x7 x8 x9 x10)  ///
      ,llfile(weib_ll)                         ///
      othervars(_t _d) othervarnames(t d)
```

A Weibull model using mlad

Python file: weib_ll.py

```
import jax.numpy as jnp
import mladutil as mu

def python_ll(beta,X,wt,M):
  lnlam =  mu.linpred(beta,X,1)
  lngam  = mu.linpred(beta,X,2)
  gam = jnp.exp(lngam)

  return(jnp.sum(M["d"]*(lnlam + lngam + (gam - 1)*jnp.log(M["t"])) -
         jnp.exp(lnlam)*M["t"]**(gam)))
```

Using mlad

```
mlad  (lnlam: x1 x2 x3 x4 x5 x6 x7 x8 x9 x10)  ///
      (lngam: x1 x2 x3 x4 x5 x6 x7 x8 x9 x10)  ///
      ,llfile(weib_ll)                         ///
      othervars(_t _d) othervarnames(t d)
```

This is a d2 evaluator!!

Weibull Model Compare Times

- Simulate 10 Million observations, 10 covariates for each linear predictor.
- Parameter estimates and standard errors identical.

Method	Time	Calls to program
Stata `ml d0`	8762 seconds	3208

Weibull Model Compare Times

- Simulate 10 Million observations, 10 covariates for each linear predictor.
- Parameter estimates and standard errors identical.

Method	Time	Calls to program
Stata `ml d0`	8762 seconds	3208
Stata `mlad`	49 seconds	22

Weibull Model Compare Times

- Simulate 10 Million observations, 10 covariates for each linear predictor.
- Parameter estimates and standard errors identical.

Method	Time	Calls to program
Stata `ml d0`	8762 seconds	3208
Stata `mlad`	49 seconds	22
Stata `ml d2`	136 seconds	22

Weibull Model Compare Times

- Simulate 10 Million observations, 10 covariates for each linear predictor.
- Parameter estimates and standard errors identical.

Method	Time	Calls to program
Stata `ml d0`	8762 seconds	3208
Stata `mlad`	49 seconds	22
Stata `ml d2`	136 seconds	22
Stata `ml lf0`	281 seconds	79

Weibull Model Compare Times

- Simulate 10 Million observations, 10 covariates for each linear predictor.
- Parameter estimates and standard errors identical.

Method	Time	Calls to program
Stata `ml d0`	8762 seconds	3208
Stata `mlad`	49 seconds	22
Stata `ml d2`	136 seconds	22
Stata `ml lf0`	281 seconds	79
Stata `streg`	207 seconds	33

More Examples

- I have tried various examples to test the speed improvements and will describe some of these below.
- All run on my desktop (8 processors - 2 threads per core).

Example 1: Interval censoring

- Stata now has `stintreg` to fit models to interval censored data.
- Implemented as `lf0`; some potential to improve speed.

Python file: `weib_ic_ll`

```
import jax.numpy as jnp
import mladutil as mu

def python_ll(beta, X, wt, M):
  lam = jnp.exp(mu.linpred(beta,X,1))
  gam = jnp.exp(mu.linpred(beta,X,2))

  lli = (jnp.where(M["ctype"]==1,jnp.log(mu.weibdens(M["ltime"],lam,gam)),0)                                    +
    jnp.where(M["ctype"]==2,jnp.log(mu.weibsurv(M["ltime"],lam,gam)),0)                                        +
    jnp.where(M["ctype"]==3,jnp.log(1 - mu.weibsurv(M["rtime"],lam,gam)),0)                                    +
    jnp.where(M["ctype"]==4,jnp.log(mu.weibsurv(M["ltime"],lam,gam)-mu.weibsurv(M["rtime"],lam,gam)),0))
    return(jnp.sum(wt*lli))
```

Stata

```
mlad (ln_lambda: = ‘covlist’, ) (lngamma: = ‘gcovlist’),
     othervars(ctype ltime rtime) llfile(weib_ic_ll)
```

Interval censoring: Compare Speeds

- 10 covariate for log(λ)
- 2 covariates for log(γ)

Sample Size	`mlad`	`stintreg`	% improvement
1,000	0.7	0.1	-474.2
10,000	0.7	0.5	-42.5
50,000	0.9	2.4	61.7
100,000	1.3	4.4	71.2
250,000	2.1	11.8	81.9
500,000	3.8	26.5	85.9
1,000,000	6.8	50.9	86.7
5,000,000	31.1	241.3	87.1
10,000,000	59.7	450.0	86.7

Interval censoring: Compare Speeds

- 10 covariate for log(λ)
- 2 covariates for log(γ)

Sample Size	`mlad`	`stintreg`	% improvement
1,000	0.7	0.1	-474.2
10,000	0.7	0.5	-42.5
50,000	0.9	2.4	61.7
100,000	1.3	4.4	71.2
250,000	2.1	11.8	81.9
500,000	3.8	26.5	85.9
1,000,000	6.8	50.9	86.7
5,000,000	31.1	241.3	87.1
10,000,000	59.7	450.0	86.7

Example 2: Cure Models

- My first Stata command was for cure models in the relative survival framework, `strsmix`.

$$S(t) = S^*(t)\left[\pi + (1-\pi)S_u(t)\right]$$

- This was an `lf0` command.

Python likelihood file

```
import jax.numpy as jnp
import mladutil as mu

def python_ll(beta, X, wt, M):
  pi  = mu.invlogit(mu.linpred(beta1,X,1))
  lam = jnp.exp(mu.linpred(beta2,X,2))
  gam = jnp.exp(mu.linpred(beta3,X,3))

  ftc = mu.weibdens(t,lam,gam)
  Stc = mu.weibsurv(t,lam,gam)
  ht = (1-pi)*(ftc)/(pi + (1-pi)*(Stc))
  St = (pi + (1-pi)*(Stc))

  return(jnp.sum(wt*(d*jnp.log(rate + ht) + jnp.log(St))))
```

Cure Models: Compare Speeds

- 10 covariates for cure proportion and Weibull λ parameter, constant γ parameter.

Sample Size	`mlad`	`strsmix`	% improvement
1,000	1.0	0.3	-208.9
10,000	1.0	2.3	55.0
50,000	1.6	8.1	79.6
100,000	2.4	17.8	86.7
250,000	4.4	43.6	89.8
500,000	8.5	84.1	89.9
1,000,000	15.1	160.7	90.6
2,500,000	40.4	444.2	90.9
5,000,000	75.3	838.0	91.0
10,000,000	150.6	1730.4	91.3

Cure Models: Compare Speeds

- 10 covariates for cure proportion and Weibull λ parameter, constant γ parameter.

Sample Size	`mlad`	`strsmix`	% improvement
1,000	1.0	0.3	-208.9
10,000	1.0	2.3	55.0
50,000	1.6	8.1	79.6
100,000	2.4	17.8	86.7
250,000	4.4	43.6	89.8
500,000	8.5	84.1	89.9
1,000,000	15.1	160.7	90.6
2,500,000	40.4	444.2	90.9
5,000,000	75.3	838.0	91.0
10,000,000	150.6	1730.4	91.3

Example 3: Splines for the log hazard function

- We fit models of the following form,

$$\ln\left[h(t)\right] = \ln\left[s(ln(t)|\gamma, k_0)\right] + \beta X$$

- $s(ln(t)|\gamma, k_0)$ is a restricted cubic spline function.
- The log-likelihood requires numerical integration

$$\ell_i = d_i \ln\left[h(t_i)\right] - \int_{t_{0i}}^{t_i} h(u)du$$

- Can be fitted in Stata using `stgenreg` (`d0`), `strcs` (`gf2`), `merlin` (`gf2`)

Splines on the log hazard scale

Python likelihood file

```
import jax.numpy as jnp
from   jax import vmap
import mladutil as mu

def python_ll(beta,X,wt,M,Nnodes):
  ## Parameters
  xb    = mu.linpred(beta,X,1)
  xbrcs = mu.linpred(beta,X,2)

  ## hazard function
  def rcshaz(t):
    vrcsgen = vmap(mu.rcsgen_beta,(0,None,None))
    return(jnp.exp(vrcsgen(jnp.log(t),M["knots"][0],beta2) + xb))

  ## cumulative hazard
  cumhaz = mu.vecquad_gl(rcshaz,M["t0"],M["t"],Nnodes,())

  return(jnp.sum(wt*(M["d"]*(xb + xbrcs) - cumhaz)))
```

Splines on the log hazard scale: Times

- Proportional hazards model, 10 covariates.
- Restricted cubic spline with 6 knots for log baseline hazard.

Sample Size	`mlad`	`stgenreg`	`strcs`	`merlin`
1,000	0.6	4.9 (87.8%)	0.4 (-50%)	1.9 (68.4%)
10,000	0.9	48 (98.1%)	2.4 (62.5%)	11 (91.7%)
50,000	2.2	193 (98.9%)	12 (82.0%)	86 (97.5%)
100,000	3.4	452 (99.2%)	27 (87.2%)	178 (98.1%)
250,000	7.4	1,125 (99.3%)	69 (89.2%)	441 (98.3%)
500,000	14.2	2,329 (99.4%)	139 (89.8%)	898 (98.4%)
1,000,000	26.4	4,694 (99.4%)	285 (90.7%)	1789 (98.5%)
2,500,000	65.0	-	678 (90.7%)	4734 (98.6%)

Splines on the log hazard scale: Times

- Proportional hazards model, 10 covariates.
- Restricted cubic spline with 6 knots for log baseline hazard.

Sample Size	`mlad`	`stgenreg`	`strcs`	`merlin`
1,000	0.6	4.9 (87.8%)	0.4 (-50%)	1.9 (68.4%)
10,000	0.9	48 (98.1%)	2.4 (62.5%)	11 (91.7%)
50,000	2.2	193 (98.9%)	12 (82.0%)	86 (97.5%)
100,000	3.4	452 (99.2%)	27 (87.2%)	178 (98.1%)
250,000	7.4	1,125 (99.3%)	69 (89.2%)	441 (98.3%)
500,000	14.2	2,329 (99.4%)	139 (89.8%)	898 (98.4%)
1,000,000	26.4	4,694 (99.4%)	285 (90.7%)	1789 (98.5%)
2,500,000	65.0	-	678 (90.7%)	4734 (98.6%)

Splines on the log hazard scale: Times

- Proportional hazards model, 10 covariates.
- Restricted cubic spline with 6 knots for log baseline hazard.

Sample Size	`mlad`	`stgenreg`	`strcs`	`merlin`
1,000	0.6	4.9 (87.8%)	0.4 (-50%)	1.9 (68.4%)
10,000	0.9	48 (98.1%)	2.4 (62.5%)	11 (91.7%)
50,000	2.2	193 (98.9%)	12 (82.0%)	86 (97.5%)
100,000	3.4	452 (99.2%)	27 (87.2%)	178 (98.1%)
250,000	7.4	1,125 (99.3%)	69 (89.2%)	441 (98.3%)
500,000	14.2	2,329 (99.4%)	139 (89.8%)	898 (98.4%)
1,000,000	26.4	4,694 (99.4%)	285 (90.7%)	1789 (98.5%)
2,500,000	65.0	-	678 (90.7%)	4734 (98.6%)

- Python code is inefficient as spline function calculated at every node every time function is called.
- See example on my website for calculating these once using the `pysetup()` option.

Example 4: Random effect models

- Random effects models can be slow as they need to perform numerical integration.
- For a flexible parametric survival model

$$\ln\left[H(t|X,Z)\right] = s(\ln(t)|\gamma, k_0) + X\beta + Zu$$

$$u \sim N(0, \Sigma_u)$$

- These models can be fitted in Stata using `stmixed`, which calls `merlin` using a `gf0`.

Example 4: Random effect models

- Random effects models can be slow as they need to perform numerical integration.
- For a flexible parametric survival model

$$\ln\left[H(t|X,Z)\right] = s(\ln(t)|\gamma, k_0) + X\beta + Zu$$

$$u \sim N(0, \Sigma_u)$$

- These models can be fitted in Stata using `stmixed`, which calls `merlin` using a `gf0`.
- I have just implemented a "proof of concept" with a random intercept and 1 covariate with random coefficent with unstructured covariance matrix.
- Currently only non-adaptive quadrature is implemented.

Random effect models

- Use setup file to pre-calculate grid of nodes and weights for numerical integration.

Python likelihood file

```
import jax.numpy as jnp
from scipy.special import roots_hermite

def mlad_setup(M):
  # Design matrix for random effects
  M['Z'] = jnp.hstack((jnp.ones((M["z1"].shape[0],1)),M["z1"]))

  # Nodes and weights
  nodes, weights = roots_hermite(M["Nnodes"])
  allnodes = jnp.repeat(jnp.asarray(jnp.sqrt(2)*nodes[:,None]),2,axis=1).T
  M['nodes'] = (jnp.asarray(jnp.meshgrid(allnodes[0,:],allnodes[1,:])).T.reshape(-1, 2))
  M['nodes'] = jnp.asarray(M['nodes'])[:,:,None]
  allweights = jnp.repeat(jnp.asarray(weights[:,None])/jnp.sqrt(jnp.pi),2,axis=1).T
  weightscomb = (jnp.asarray(jnp.meshgrid(allweights[0,:],allweights[1,:])).T.reshape(-1, 2))
  M['weights'] = jnp.prod(weightscomb,axis=1)
  return(M)
```

Random effect models

Python likelihood file

```
import jax.numpy as jnp
import mladutil as mu
from jax import vmap
from jax.numpy.linalg import cholesky

def python_ll(beta,X,wt,M,Nid):
  xb      =  mu.linpred(beta,X,1)
  dxb     =  mu.linpred(beta,X,2)
  sigma0  =  mu.linpred(beta,X,3)[0,0]
  sigma1  =  mu.linpred(beta,X,4)[0,0]
  sigma01 =  mu.linpred(beta,X,5)[0,0]

  V = jnp.vstack((jnp.hstack((sigma0, sigma01)),
  jnp.hstack((sigma01,sigma1))))
  C = cholesky(V)

  def calc_lnft_fpm(v):
    lp = xb + M["Z"]@C@v
    return((M["d"]*(jnp.log(dxb) + lp) - jnp.exp(lp))[:,0])

  vect_calc_lnft_fpm = vmap(calc_lnft_fpm,(0),1)

  def getllj(v):
    logF = vect_calc_lnft_fpm(v)
    return(jnp.exp(mu.sumoverid(M["id"],logF,Nid)))

  llj = jnp.log(jnp.sum(M["weights"]*getllj(M["nodes"]),axis=1,keepdims=True))
  return(jnp.sum(llj))
```

Random effect models: Times

- Flexible parametric survival model with random intercept, random coefficient and unstructured covariance matrix.
- Uses non-adaptive quadrature.
- Estimates are identical

Sample Size	`mlad`	`stmixed`	% improvement
1,000	2.3	4.9	53.6%
10,000	2.6	74.0	96.5%
50,000	4.2	250.1	98.3%
100,000	7.4	403.1	98.2%
250,000	15.2	1,166.4	98.7%
500,000	29.4	2,115.5	98.6%
1,000,000	59.2	4,274.9	98.6%
1,500,000	83.9	9,548.8	99.1%

- Speed comes at a cost of being memory hungry.

Random effect models: Times

- Flexible parametric survival model with random intercept, random coefficient and unstructured covariance matrix.
- Uses non-adaptive quadrature.
- Estimates are identical

Sample Size	`mlad`	`stmixed`	% improvement
1,000	2.3	4.9	53.6%
10,000	2.6	74.0	96.5%
50,000	4.2	250.1	98.3%
100,000	7.4	403.1	98.2%
250,000	15.2	1,166.4	98.7%
500,000	29.4	2,115.5	98.6%
1,000,000	59.2	4,274.9	98.6%
1,500,000	83.9	9,548.8	99.1%

- Speed comes at a cost of being memory hungry.

Official commands

- Speed gains for official Stata commands implemented with `d2`, `lf2`, `gf2` may be not as dramatic.
- However, still some moderate improvements.

Official commands

- Speed gains for official Stata commands implemented with `d2`, `lf2`, `gf2` may be not as dramatic.
- However, still some moderate improvements.
- We sometimes use Poisson regression to model rates - through splitting of time-scales into intervals.
- This can lead to large datasets as many rows for each individual.

Python: poisson.py

```
import jax.numpy as jnp
import mladutil as mu

def python_ll(beta, X, wt, M):
  xb = mu.linpred(beta,X,1)
  return(jnp.sum(wt*(M["y"]*xb - jnp.exp(xb))))
```

Stata code

```
mlad (xb:  = ‘covlist’, offset(lnrisktime)) ///
     , othervars(y) llfile(poisson)
```

glm

Individuals	rows	`mlad`-1P	`glm`-1P	`glm`-2P
1,000	41,297	0.3	0.7 (57.1%)	0.5 (40.0%)
10,000	410,119	1.8	6.8 (73.5%)	4.3 (58.1%)
50,000	2,040,827	6.6	31.1 (78.8)	18.7 (30.5%)
100,000	4,078,914	13.0	63.4 (79.5%)	36.8 (64.7%)
250,000	10,185,191	43.0	223.8 (80.7%)	143.6 (70.1%)
500,000	20,397,607	72.2	393.6 (81.6%)	253.8 (71.6%)

glm

Individuals	rows	mlad-1P	glm-1P	glm-2P
1,000	41,297	0.3	0.7 (57.1%)	0.5 (40.0%)
10,000	410,119	1.8	6.8 (73.5%)	4.3 (58.1%)
50,000	2,040,827	6.6	31.1 (78.8)	18.7 (30.5%)
100,000	4,078,914	13.0	63.4 (79.5%)	36.8 (64.7%)
250,000	10,185,191	43.0	223.8 (80.7%)	143.6 (70.1%)
500,000	20,397,607	72.2	393.6 (81.6%)	253.8 (71.6%)

glm

Individuals	rows	mlad-1P	glm-1P	glm-2P
1,000	41,297	0.3	0.7 (57.1%)	0.5 (40.0%)
10,000	410,119	1.8	6.8 (73.5%)	4.3 (58.1%)
50,000	2,040,827	6.6	31.1 (78.8)	18.7 (30.5%)
100,000	4,078,914	13.0	63.4 (79.5%)	36.8 (64.7%)
250,000	10,185,191	43.0	223.8 (80.7%)	143.6 (70.1%)
500,000	20,397,607	72.2	393.6 (81.6%)	253.8 (71.6%)

- Can post estimates to glm to use post-estimation commands.

Implementation

- `ml` and `mlad` will be called from "user friendly" programs.
- Introduce a `python` option and syntax is similar to something like

Python likelihood file

```
if "`python'" == "" {
  ml ......
}
else {
  mlad ......
}
```

- This is how it is implemented in `strcs` - updated version coming soon.

Summary

- Important speed improvements for official and user written commands.
- Developers of estimation command can easily add to their existing commands to get important speed gains.
- Benefits of multiple CPUs, without using Stata MP.
- Automatic differentiation simplifies development.
- Started to add to our own commands (`strcs`).

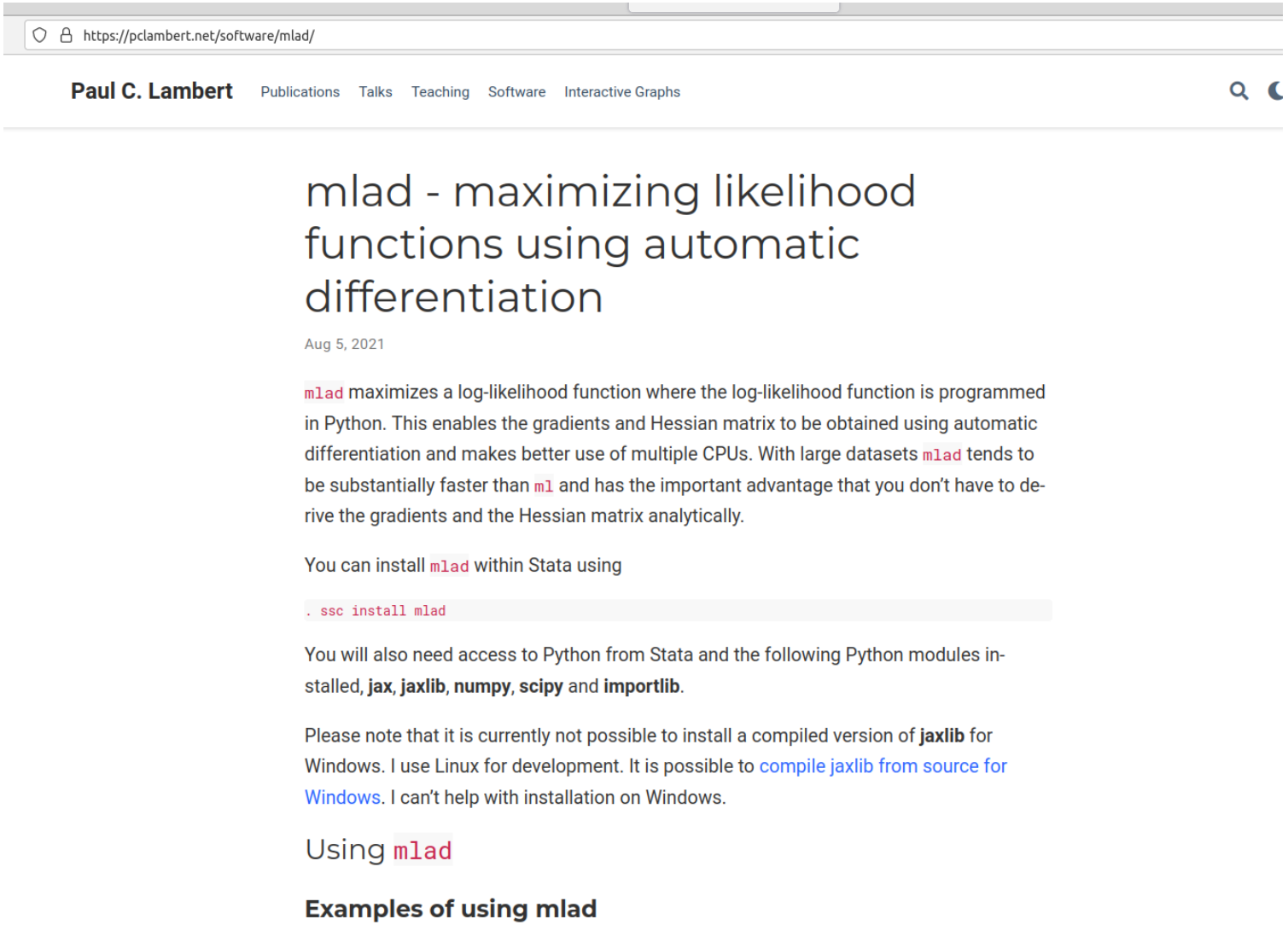
Other extensions

- If weights are specified, automatically passed to Python.
- If offsets are specified, automatically incorporated into linear predictor.
- Allows factors variables (slightly slower).
- Allows constraints.
- Allows robust and cluster robust standard errors (Scores obtained using AD).
- Can choose to write gradient and Hessian function in Python.

Issues

- Not worth it for small sample sizes.
- Memory intensive for numerical integration (using `vmap()`)
- User needs to install Python and `jax` and `jaxlib` Python modules.
- Less support for Windows (need to compile `jaxlib` from source).
- Python code needs to follow certain style for fast jit compilation.

Examples on my Website



`www.pclambert.net/software/mlad`