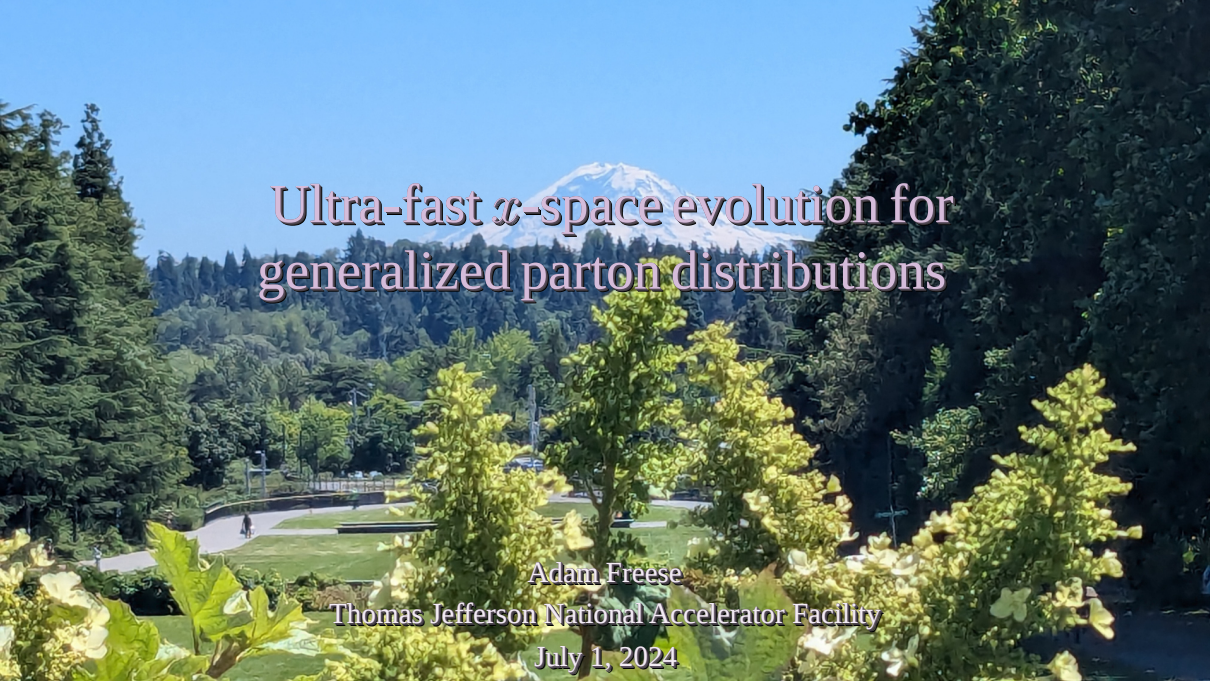
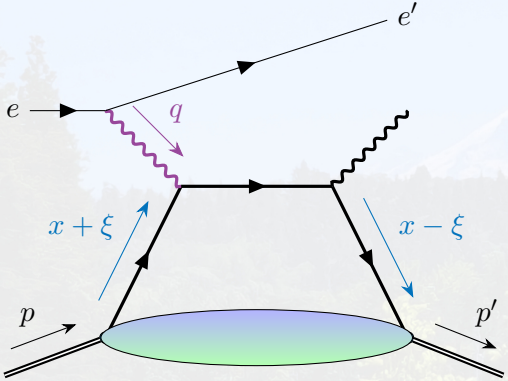


Ultra-fast x-space evolution for generalized parton distributions

Adam Freese
Thomas Jefferson National Accelerator Facility

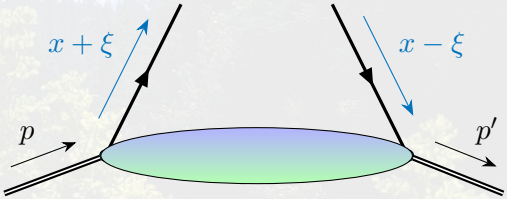
July 1, 2024

Generalized parton distributions



Deeply virtual Compton scattering

$\mathcal{H}(\xi, t; Q^2)$

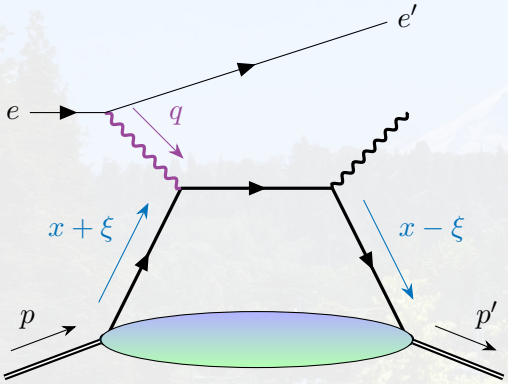


Generalized parton distribution

$H(x, \xi, t; Q^2)$

- **Generalized parton distributions** are 4-variable functions.
- Probed in processes such as **deeply virtual Compton scattering** (DVCS).

The GPD variables



$$x = \frac{(k+k')\cdot n}{(p+p')\cdot n}$$

$$\xi = \frac{(p-p')\cdot n}{(p+p')\cdot n}$$

$$t = (p'-p)^2$$

$$Q^2 = -q^2$$

n defines the reference frame

- x is *average* momentum fraction of struck parton.
- 2ξ is the **skewness**: momentum fraction lost by struck parton.
- t is the invariant momentum transfer.
- GPDs also depend on **resolution scale** Q^2.

Evolution equations

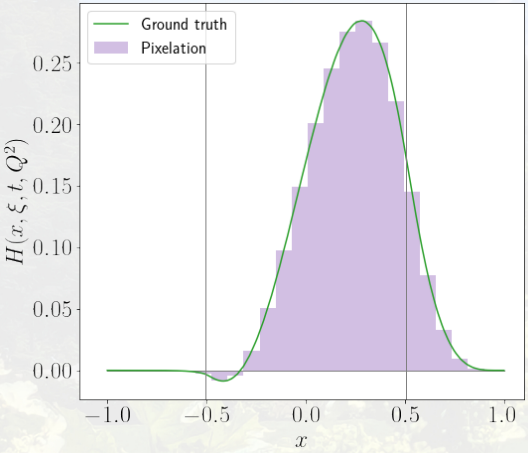
- GPDs obey **evolution equations** for Q^2 dependence:

$$\frac{\mathrm{d}H(x,\xi,t,Q^2)}{\mathrm{d}\log(Q^2)} = \int_{-1}^{+1} \mathrm{d}y\, K(x,y,\xi,Q^2) H(y,\xi,t,Q^2)$$

  - **Kernel** $K(x,y,\xi,Q^2)$ known theoretically.
- Only need 3D GPD at one scale Q_0^2 to fix 4D GPD at all Q^2.
  - **This** is what we (via neural network) parametrize.
- Need **fast** and **differentiable** code to perform evolution.

Pixelation

- GPD is **pixelated** in x-space.
- Per (ξ, t, Q^2) value is effectively column matrix.



$$H_i = \begin{bmatrix} 0 \\ -3.50688094 \times 10^{-8} \\ -2.23178870 \times 10^{-6} \\ \vdots \\ 2.93122078 \times 10^{-5} \end{bmatrix}$$

Evolution matrices

- GPD at (ξ, t, Q^2) and (ξ, t, Q_0^2) are both column matrices.
  - An $N_x \times N_x$ square matrix connects them.
  - **Evolution matrix** (or transfer matrix)
  - Solve evolution equation by constructing these matrices!
- Evolution matrices fit our needs:
  - Matrix multiplication is fast (especially with GPUs).
  - Matrix multiplication is differentiable.
  - Can easy be implemented via `torch.einsum`

$$H_i(\xi, t, Q^2) = \sum_{j=1}^{N_x} M_{ij}(\xi, Q_0^2 \to Q^2) H_j(\xi, t, Q_0^2)$$

Two code bases: PyTorch vs. Fortran

- We have two code bases for making evolution matrices, in PyTorch and Fortran.
- Can explore different algorithms & strategies.
- Different codes serve as a cross-check.
  - Ideas developed in one can also be applied to the other.

PyTorch implementation

- ☑ All operations are matrix multiplication
- ☑ Conceptually straightforward
- ☑ Runs on (and leverages) GPUs
- ☑ Still fast on CPUs
- ☒ Can’t use adaptive integration/interpolation
- ☒ Numerically noisy
- ☑ Seamlessly integrated into PyTorch codebase

Fortran implementation

- ☒ Uses some non-matrix methods
- ☒ Conceptually complicated
- ☒ CPU-only
- ☒ Slower than PyTorch code
- ☑ Leverages adaptive methods
- ☑ Numerically well-behaved
- ☑ Python wrapper allows integration into codebase


PyTorch implementation

Integral discretization

- First step is to discretize the integral:

$$S(x,\xi,t,Q^2) = \int_{-1}^{+1} \mathrm{d}y\, K(x,y,\xi,Q^2) H(y,\xi,t,Q^2)$$

- Kernel made up of three distributions; must be integrated separately:

$$K(x,y,\xi,Q^2) = K_R(x,y,\xi,Q^2) + [K_P(x,y,\xi,Q^2)]_+ + K_C(Q^2)\delta(y-x)$$

  - **Regular piece**—just a normal integral:

$$\int_{-1}^{+1} \mathrm{d}y\, K_R(x,y,\xi,Q^2) H(y,\xi,t,Q^2)$$

  - **Plus distribution piece**:

$$\int_{-1}^{+1} \mathrm{d}y\, [K_P(x,y,\xi,Q^2)]_+ H(y,\xi,t,Q^2) \equiv \int_{-1}^{+1} \mathrm{d}y\, K_P(x,y,\xi,Q^2) \Big(H(y,\xi,t,Q^2) - H(x,\xi,t,Q^2) \Big) + H(x,\xi,t,Q^2) \int_{-1}^{+1} \mathrm{d}y \Big(K_P(x,y,\xi,Q^2) - K_P(y,x,\xi,Q^2) \Big)$$

  - **Constant piece** (or delta distribution piece):

$$\int_{-1}^{+1} \mathrm{d}y\, K_C(Q^2)\delta(y-x) H(y,\xi,t,Q^2) \equiv K_C(Q^2) H(x,\xi,t,Q^2)$$

Regular piece

- Regular piece approximated using **Gauss-Legendre quadrature**:

$$S_R(x,\xi,t,Q^2) = \int_{-1}^{+1} \mathrm{d}y\, K_R(x,y,\xi,Q^2) H(y,\xi,t,Q^2)$$
$$\approx \sum_{g=1}^{N_g} w_g K_R(x,y_g,\xi,Q^2) H(y_g,\xi,t,Q^2)$$

  - y_g are roots of N_gth order Legendre polynomial.
  - w_g are Gaussian weights at these roots.
  - Need $N_g \sim 1000$ for good accuracy.

cubic-turbo interpolation

- Quadrature grid and pixelation grid are not the same.
  - Must interpolate to quadrature grid.
- Use **cubic-turbo** method by Daniel Adamiak.
  - Modified cubic Hermite polynomials (except at endpoints).
  - "Modified": numerical derivative computed using values at adjacent points.
  - Ordinary cubic interpolation used for endpoints.
  - Parallelized code leverages GPUs for massive speedup—hence "turbo".
- Interpolation done via matrix multiplication:

$$H(y_g, \xi, t, Q^2) = \sum_{j=1}^{N_x} L_{gj} H(y_j, \xi, t, Q^2)$$

  - **Interpolation matrix** L_{gj} constructed via **cubic-turbo**.

Regular piece: matrix formulation

- Using cubic-turbo and Gauss-Legendre quadrature:

$$S_R(x_i,\xi,t,Q^2) \approx \sum_{j=1}^{N_x} \underbrace{\left(\sum_{g=1}^{N_g} g_w K_R(x_i,y_g,\xi,Q^2) L_{gj}\right)}_{\left(K_R(\xi,Q^2)\right)_{ij}} \overbrace{H(y_j,\xi,t,Q^2)}^{H_j(\xi,t,Q^2)}$$

- Right-hand side is now matrix multiplication:

$$S_R(x_i,\xi,t,Q^2) \approx \sum_{j=1}^{N_x} \left(K_R(\xi,Q^2)\right)_{ij} H_j(\xi,t,Q^2)$$

- The matrix $\left(K_R(\xi,Q^2)\right)_{ij}$ is *independent of the GPD*.
  - Can be computed once, stored in memory.
  - Doesn't need to be re-computed for each trial GPD during fit/training/etc.

Plus distribution piece

- Plus distribution piece is a sum of two integrals:

$$S_P(x,\xi,t,Q^2) \equiv \int_{-1}^{+1} \mathrm{d}y\, [K_P(x,y,\xi,Q^2)]_+ H(y,\xi,t,Q^2) = S_P^{(1)}(x,\xi,t,Q^2) + S_P^{(2)}(x,\xi,t,Q^2)$$

$$S_P^{(1)}(x,\xi,t,Q^2) = \int_{-1}^{+1} \mathrm{d}y\, K_P(x,y,\xi,Q^2)\Big(H(y,\xi,t,Q^2) - H(x,\xi,t,Q^2)\Big)$$

$$S_P^{(2)}(x,\xi,t,Q^2) = H(x,\xi,t,Q^2) \int_{-1}^{+1} \mathrm{d}y \left(K_P(x,y,\xi,Q^2) - K_P(y,x,\xi,Q^2)\right)$$

- Presents numerical difficulties because of $1/(y-x)$ factors in K_P.

Plus distribution piece: first integral

- Do first integral using Gauss-Legendre quadrature and cubic-turbo:

$$S_P^{(1)}(x_i,\xi,t,Q^2) = \int_{-1}^{+1} \mathrm{d}y\, K_P(x_i,y,\xi,Q^2)\Big(H(y,\xi,t,Q^2) - H(x_i,\xi,t,Q^2)\Big)$$
$$\approx \sum_{g=1}^{N_g} w_g K_P(x_i,y_g,\xi,Q^2)\left(\sum_{j=1}^{N_x} L_{gj} H(y_j,\xi,t,Q^2) - H(x_i,\xi,t,Q^2)\right)$$

- **Matrix implementation**:

$$S_P^{(1)}(x_i,\xi,t,Q^2) \approx \sum_{j=1}^{N_x} \underbrace{\left(\sum_{g=1}^{N_g} w_g K_P(x_i,y_g,\xi,Q^2)\Big[L_{gj} - \delta_{ij}\Big]\right)}_{\left(K_P^{(1)}(\xi,Q^2)\right)_{ij}} H_j(\xi,t,Q^2)$$

- Current implementation numerically noisy.

Plus distribution piece: second integral

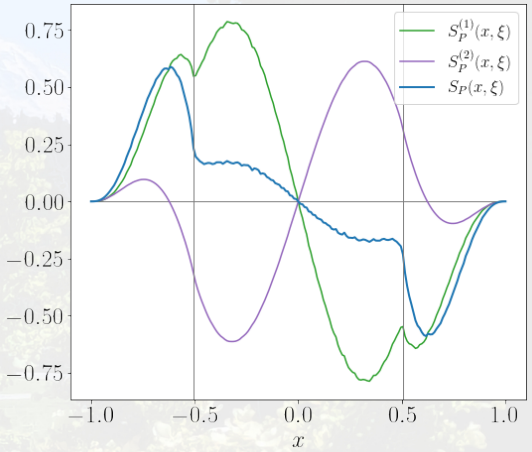
- Second integral gives diagonal matrix:

$$S_P^{(2)}(x_i,\xi,t,Q^2) = \sum_{j=1}^{N_x} \underbrace{\left(\int_{-1}^{+1} \mathrm{d}y \left(K_P(x_i,y,\xi,Q^2) - K_P(y,x_i,\xi,Q^2)\right)\right)\delta_{ij}}_{\left(K_P^{(2)}(\xi,Q^2)\right)_{ij}} H_j(\xi,t,Q^2)$$

- Current PyTorch implementation does integral with `torch.trapz`
  - Surprisingly smooth result, despite singularity at $y = x$.
  - Numerical issues for $x \sim \xi$; fixed by interpolating from adjacent points.
- Alternate Fortran implementation uses adaptive integration—more accurate result.
- Could do integral analytically (only feasible at leading order).

Numerical noise in current implementation

- Numerical noise in $S_P^{(1)}$.
  - The term that integrates $H(y) - H(x)$ …
  - …and has $1/(y - x)$ in the integrand.
- Cause unclear.
- Noise not present in Fortran code.
- Noise disappears in overall solution.
  - Maybe don't worry about it?
- **Suggestions welcome**



Constant piece

- The constant piece (delta distribution piece) is trivial.

$$
\begin{aligned}
S_C(x_i, \xi, t, Q^2) &= \int_{-1}^{+1} \mathrm{d}y \, K_C(Q^2) \delta(y - x_i) H(y, \xi, t, Q^2) \\
&= \sum_{j=1}^{N_x} \underbrace{\Big(\delta_{ij} K_C(Q^2)\Big)}_{\big(K_C(Q^2)\big)_{ij}} H_j(\xi, t, Q^2)
\end{aligned}
$$

Fortran implementation

Regular piece

- Regular piece approximated using **Gauss-Kronrod quadrature**.
  - The domain $[-1, 1]$ is broken into **six pieces** with boundaries:

$$-1 < \min(-\xi, -|x|) < \max(-\xi, -|x|) < 0 < \min(\xi, |x|) < \max(\xi, |x|) < 1$$

  - x and ξ grids must be misaligned.
  - Interpolation done differently for **every x and ξ point**.
  - 15-point quadrature used inside each region.

$$S_R(x, \xi, t, Q^2) \approx \sum_{g=1}^{N_g=6\times 15} w_g K_R(x, y_g, \xi, Q^2) H(y_g, \xi, t, Q^2)$$

$$\approx \sum_{j=1}^{N_x} \underbrace{\left(\sum_{g=1}^{N_g} w_g K_R(x_i, y_g, \xi, Q^2) L_{gj}(x_i, \xi) \right)}_{\left(K_R(\xi, Q^2)\right)_{ij}} H_j(\xi, t, Q^2)$$

- I use (piecewise) sixth-order Newton polynomials to interpolate.

Plus distribution piece

- Reminder: plus distribution piece is a sum of two integrals:

$$S_P(x,\xi,t,Q^2) \equiv \int_{-1}^{+1} \mathrm{d}y\, [K_P(x,y,\xi,Q^2)]_+ H(y,\xi,t,Q^2) = S_P^{(1)}(x,\xi,t,Q^2) + S_P^{(2)}(x,\xi,t,Q^2)$$

$$S_P^{(1)}(x,\xi,t,Q^2) = \int_{-1}^{+1} \mathrm{d}y\, K_P(x,y,\xi,Q^2)\Big(H(y,\xi,t,Q^2) - H(x,\xi,t,Q^2)\Big)$$

$$S_P^{(2)}(x,\xi,t,Q^2) = H(x,\xi,t,Q^2) \int_{-1}^{+1} \mathrm{d}y \left(K_P(x,y,\xi,Q^2) - K_P(y,x,\xi,Q^2)\right)$$

- Still presents numerical difficulties because of $1/(y-x)$ factors in K_P.

Plus distribution piece: first integral

- Do first integral via Gauss-Kronrod rule still.
  - Break into same six integration regions.
  - Use same sixth-order Newton interpolation.
- **Matrix implementation**:

$$S_P^{(1)}(x_i,\xi,t,Q^2) \approx \sum_{j=1}^{N_x} \underbrace{\left(\sum_{g=1}^{N_g} w_g K_P(x_i,y_g,\xi,Q^2)\Big[L_{gj}(x_i,\xi)-\delta_{ij}\Big]\right)}_{\left(K_P^{(1)}(\xi,Q^2)\right)_{ij}} H_j(\xi,t,Q^2)$$

- The Fortran implentation is *not noisy*.

Plus distribution piece: second integral

- Second integral gives diagonal matrix:

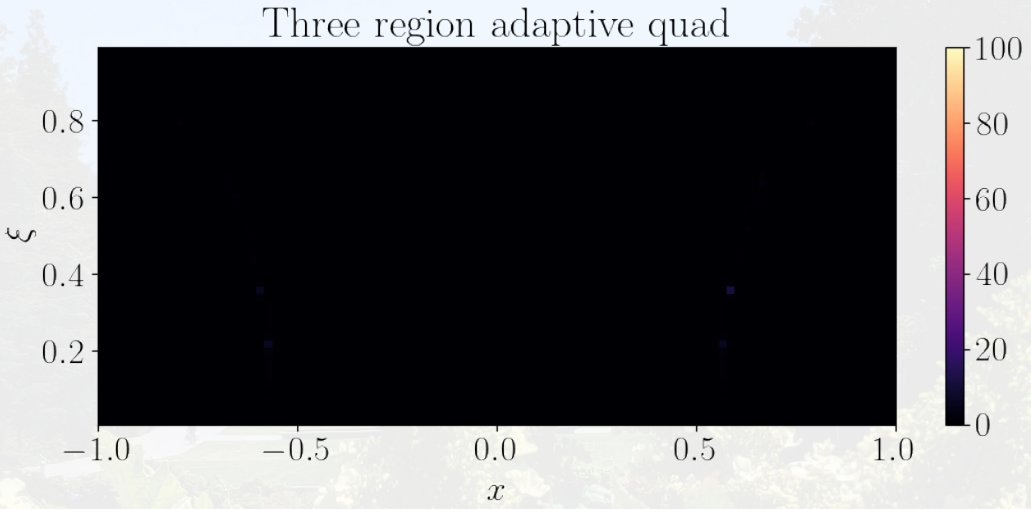
$$S_P^{(2)}(x_i, \xi, t, Q^2) = \sum_{j=1}^{N_x} \underbrace{\left(\int_{-1}^{+1} \mathrm{d}y \left(K_P(x_i, y, \xi, Q^2) - K_P(y, x_i, \xi, Q^2) \right) \right) \delta_{ij}}_{\left(K_P^{(2)}(\xi, Q^2) \right)_{ij}} H_j(\xi, t, Q^2)$$

- I get most accurate results using **adaptive quadrature** and **three regions**, with boundaries:

$$-1 < -|x| < |x| < 1$$

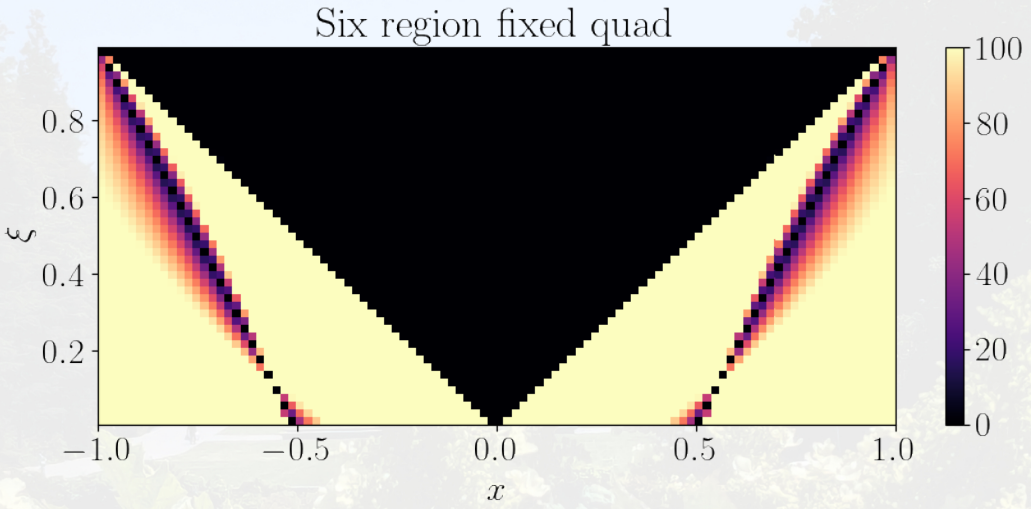
- Can get analytic results, & thus benchmark different integration methods.

Second integral: three-region method



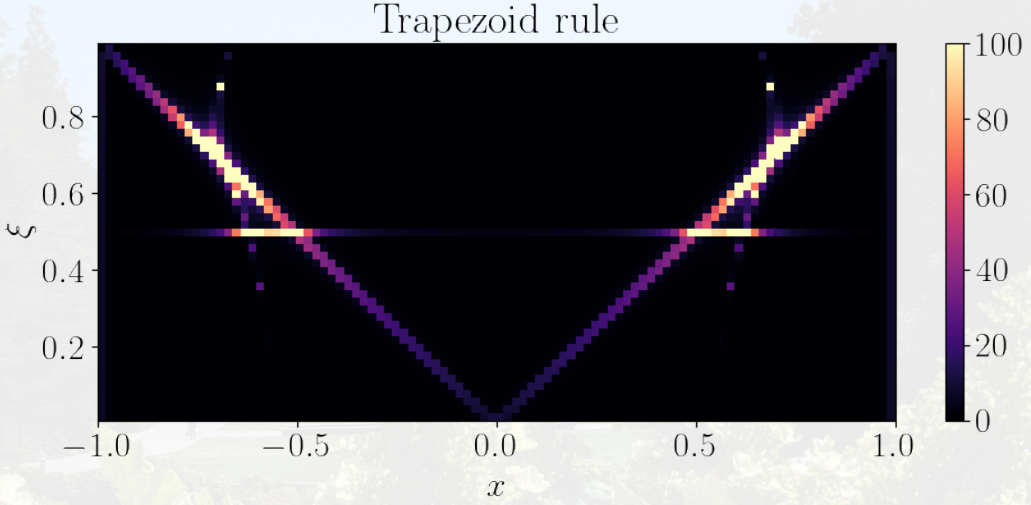
- Relative error compared to analytic result for QQ kernel.

Second integral: six-region method



- Relative error compared to analytic result for QQ kernel.

Second integral: trapezoid method



- Relative error compared to analytic result for QQ kernel.

Interpixels

- Adaptive quadrature incompatible with fixed interpolation matrices.
- **Interpixels** (**interp**olated **pixel**): interpolation basis functions.
  - Exploit linearity of Newton interpolation:

$$N[y_1 + y_2](x) = N[y_1](x) + N[y_2](x)$$

  - GPD pixelation is a sum of pixels:

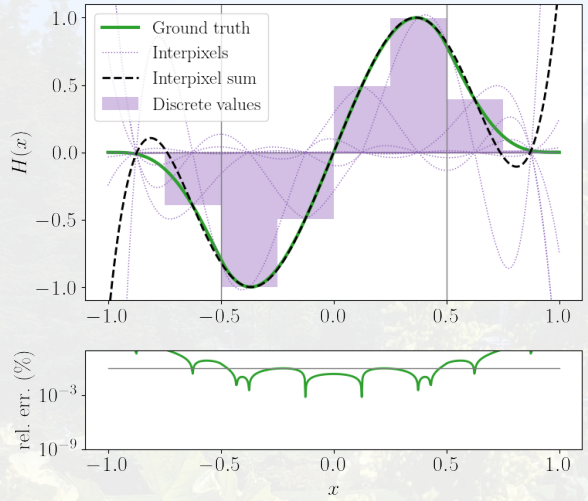
$$\boldsymbol{H} = \begin{bmatrix} h_1 \\ h_2 \\ \vdots \\ h_n \end{bmatrix} = h_1 \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + h_2 \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix} + \ldots + h_n \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix} \equiv h_1\hat{e}_1 + h_2\hat{e}_2 + \ldots + h_n\hat{e}_n$$

  - Interpolated pixelation is a sum of interpixels!

$$N[\boldsymbol{H}](x) = h_1 N[\hat{e}_1](x) + h_2 N[\hat{e}_2](x) + \ldots + h_n N[\hat{e}_n](x)$$

- Get kernel matrix by putting $H[\hat{e}_j](x)$ into integrals.

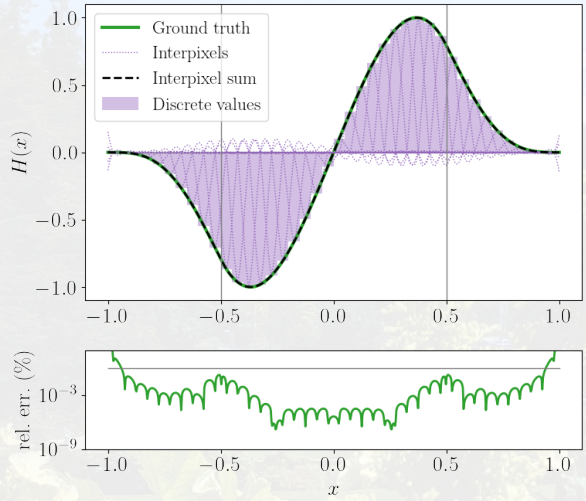
Interpixel demo



$$n_x = 8$$

- Interpixel is a *piecewise* polynomial.
  - Of fixed order.
  - Avoids Runge phenomennon.
- Knots on the discrete x grid.
- Each interpixel is oscillatory.
- Oscillations cancel in sum.
- Improvement at high N_x.

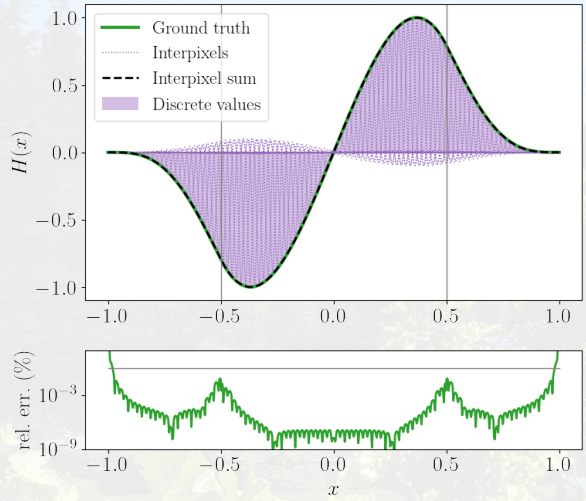
Interpixel demo



$$n_x = 40$$

- Interpixel is a *piecewise* polynomial.
  - Of fixed order.
  - Avoids Runge phenomennon.
- Knots on the discrete x grid.
- Each interpixel is oscillatory.
- Oscillations cancel in sum.
- Improvement at high N_x.

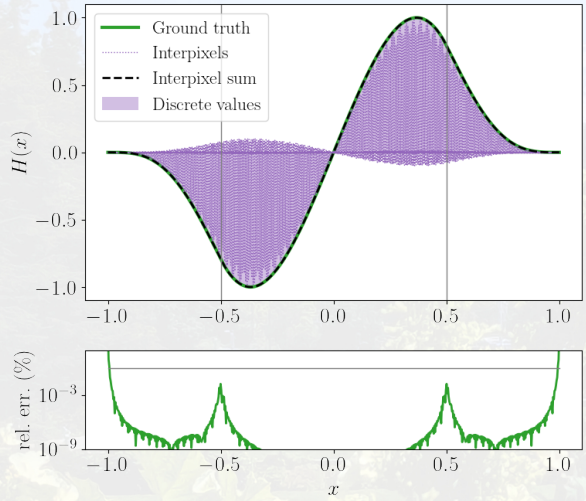
Interpixel demo



$$n_x = 100$$

- Interpixel is a *piecewise* polynomial.
  - Of fixed order.
  - Avoids Runge phenomennon.
- Knots on the discrete x grid.
- Each interpixel is oscillatory.
- Oscillations cancel in sum.
- Improvement at high N_x.

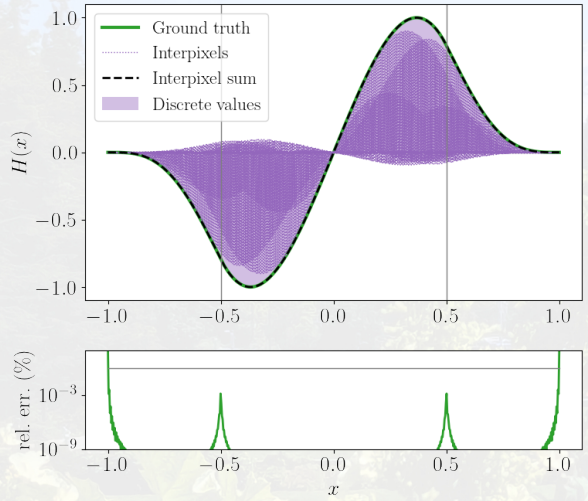
Interpixel demo



$$n_x = 300$$

- Interpixel is a *piecewise* polynomial.
  - Of fixed order.
  - Avoids Runge phenomennon.
- Knots on the discrete x grid.
- Each interpixel is oscillatory.
- Oscillations cancel in sum.
- Improvement at high N_x.

Interpixel demo



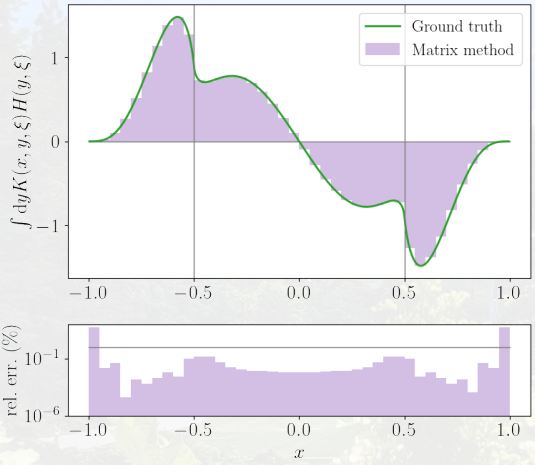
$n_x = 1000$

- Interpixel is a *piecewise* polynomial.
  - Of fixed order.
  - Avoids Runge phenomennon.
- Knots on the discrete x grid.
- Each interpixel is oscillatory.
- Oscillations cancel in sum.
- Improvement at high N_x.

Reasons for interpixels

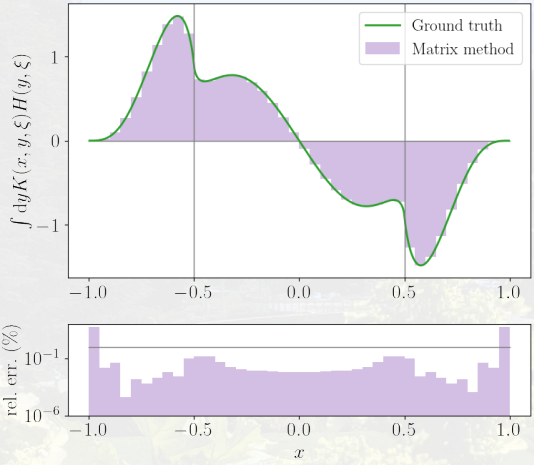
- Don’t need to store big interpolation matrices in memory.
- More flexible (adaptive or (x, ξ)-dependent) interpolation allowed.
- Allows sampling kernels arbitrarily finely in a controlled way.

Accuracy benchmark: non-singlet

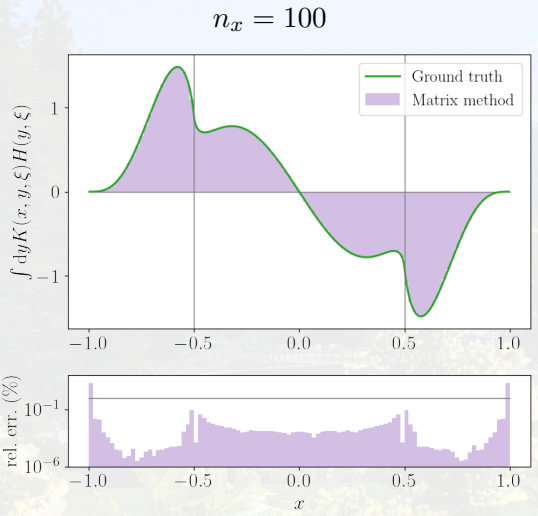


- "Ground truth" determined by adaptive integration of model function.
- Error represents error from both pixelation & interpolation.
- Sub-percent error even at $n_x = 40$.

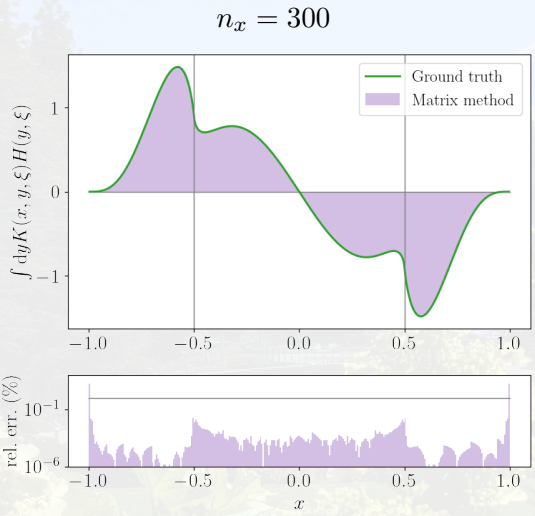
Increasing pixel density: non-singlet



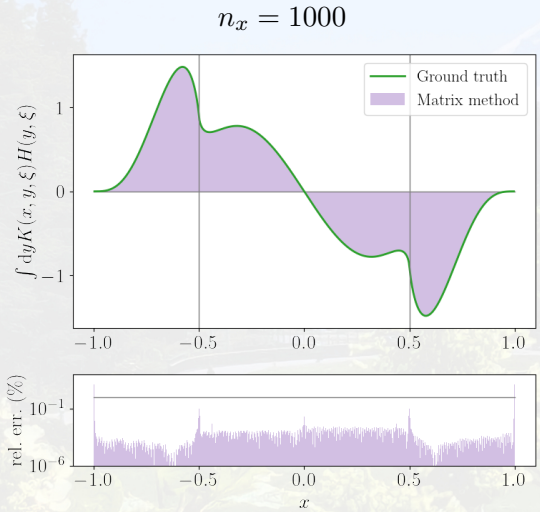
Increasing pixel density: non-singlet



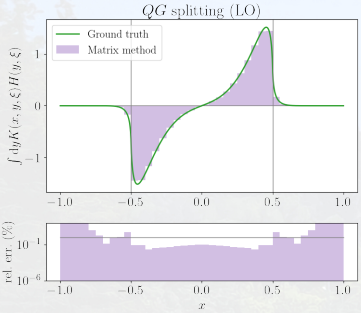
Increasing pixel density: non-singlet

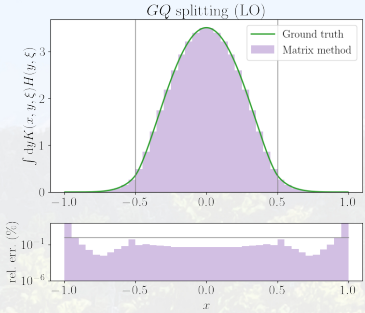


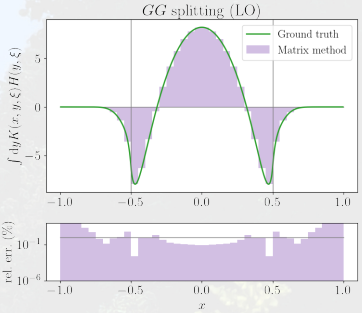
Increasing pixel density: non-singlet



Accuracy benchmark: singlet

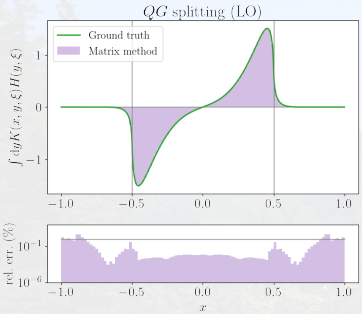


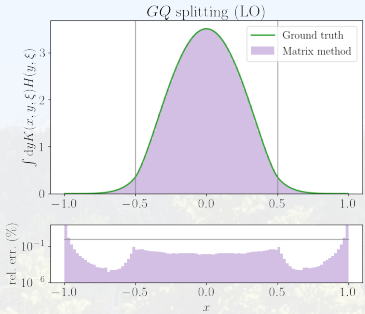


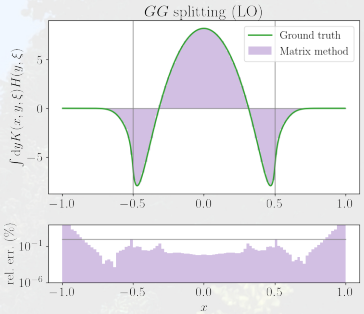


- $n_x = 40$
- Accuracy increases with pixel density.
- Seems to require more pixels than non-singlet.

Accuracy benchmark: singlet

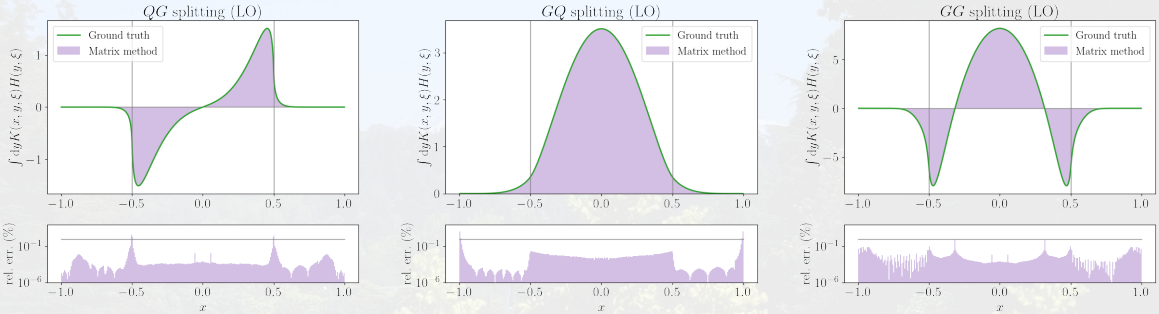






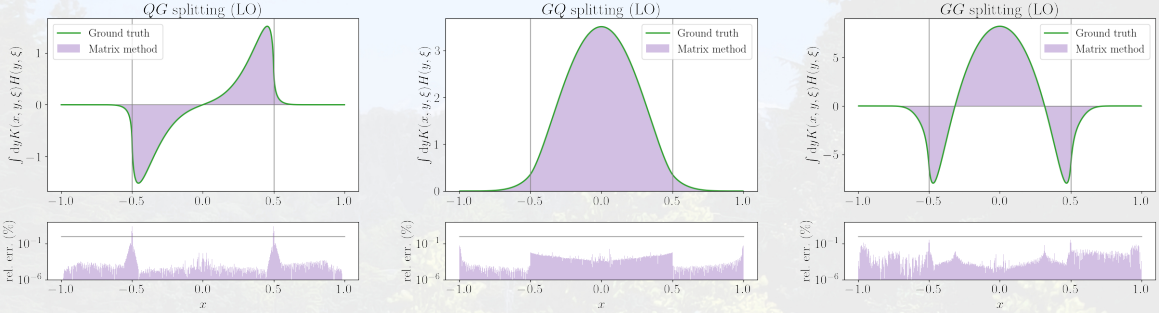
- $n_x = 100$
- Accuracy increases with pixel density.
- Seems to require more pixels than non-singlet.

Accuracy benchmark: singlet



- $n_x = 300$
- Accuracy increases with pixel density.
- Seems to require more pixels than non-singlet.

Accuracy benchmark: singlet



- $n_x = 1000$
- Accuracy increases with pixel density.
- Seems to require more pixels than non-singlet.


Solving the evolution equations

Differential matrix equation

- Combining pieces gives a matrix form of the evolution kernel:

$$K_{ij}(\xi, Q^2) = \left(K_R(\xi, Q^2)\right)_{ij} + \left(K_P^{(1)}(\xi, Q^2)\right)_{ij} + \left(K_P^{(2)}(\xi, Q^2)\right)_{ij} + \left(K_C(Q^2)\right)_{ij}$$

- Turns evolution equation into a **matrix differential equation**:

$$\frac{\mathrm{d}H_i(\xi, Q^2)}{\mathrm{d}\log(Q^2)} = \sum_{j=1}^{N_x} K_{ij}(\xi, Q^2) H_j(\xi, Q^2)$$

- This can be solved using Runge-Kutta.

Evolution matrices

- Solution to the evolution equation, via RK4:

$$H_i(\xi, t, Q^2_{\text{fin}}) = \sum_{j=1}^{N_x} M_{ij}(\xi, Q^2_{\text{ini}} \to Q^2_{\text{fin}}) H_j(\xi, Q^2_{\text{ini}})$$

- **Evolution matrix**:

$$M_{ij}(\xi, Q^2_{\text{ini}} \to Q^2_{\text{fin}}) = \delta_{ij} + \frac{1}{6} \log \frac{Q^2_{\text{fin}}}{Q^2_{\text{ini}}} \left(M^{(1)}_{ij}(\xi) + 2M^{(2)}_{ij}(\xi) + 2M^{(3)}_{ij}(\xi) + M^{(4)}_{ij}(\xi) \right)$$

- Build using RK4:

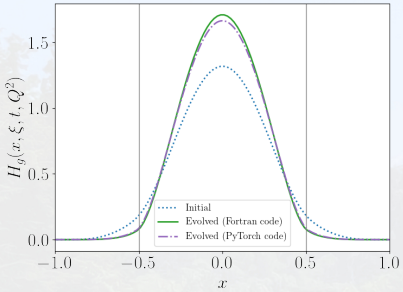
$$M^{(1)}_{ij}(\xi) = K_{ij}(\xi, Q^2_{\text{ini}})$$

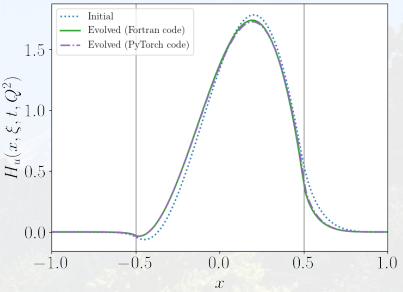
$$M^{(2)}_{ij}(\xi) = \sum_{l=1}^{N_x} K_{il}(\xi, Q^2_{\text{mid}}) \left(\delta_{lj} + \frac{1}{2} \log \frac{Q^2_{\text{fin}}}{Q^2_{\text{ini}}} M^{(1)}_{lj}(\xi) \right)$$

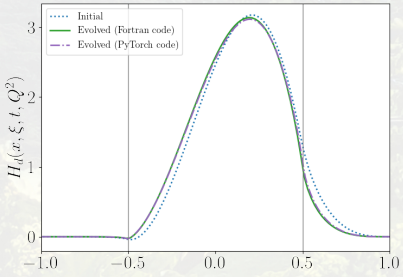
$$M^{(3)}_{ij}(\xi) = \sum_{l=1}^{N_x} K_{il}(\xi, Q^2_{\text{mid}}) \left(\delta_{lj} + \frac{1}{2} \log \frac{Q^2_{\text{fin}}}{Q^2_{\text{ini}}} M^{(2)}_{lj}(\xi) \right)$$

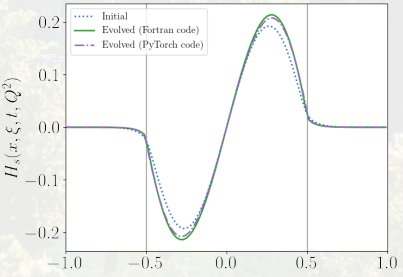
$$M^{(4)}_{ij}(\xi) = \sum_{l=1}^{N_x} K_{il}(\xi, Q^2_{\text{fin}}) \left(\delta_{lj} + \log \frac{Q^2_{\text{fin}}}{Q^2_{\text{ini}}} M^{(3)}_{lj}(\xi) \right)$$

Numerical solution









$Q_0^2 = 1\ \text{GeV}^2$

$Q^2 = 25\ \text{GeV}^2$

$t = 0$

$\xi = 0.5$

- Slight discrepancy between codes.
- Noise gone in PyTorch code?

Crude timing benchmarks

- Ran code to make evolution matrices at 10 Q^2 values from 1 GeV^2 to 25 GeV^2.
- **PyTorch code**:
  - on **GPU** (JLab farm): 10.8 s
  - on CPU (JLab farm): 19.7 s
- **Fortran code**
  - on CPU (JLab farm): 26.3 s
  - on CPU (my laptop): 54 s
- **Caveats** (comparison is not apples-to-apples):
  - PyTorch code uses $N_x = 200$ and $N_\xi = 100$. (This is hard-coded.)
  - Fortran code uses $N_x = 100$ and $N_\xi = 50$. (Segfaults at $N_x = 200$.)
  - PyTorch only computes helicity-independent kernels, $N_f = 3$.
  - Fortran computes helicity-independent & -dependent kernels, $N_f \in \{3, 4, 5\}$.
- Overall seems PyTorch code is faster.

Remaining issues

Memory issues

- Fortran RK4 solver segfaults for $n_x > 180$.
- Cause possibly from arithmetic operations on stack?
- Fails on the following line:

```
1  MV_NS(:,:,ixi,iQ2) = MV_NS(:,:,ixi,iQ2) + &
2      & rk4_NS(nx, nxi, Q2_cache(iQ2-1), Q2_cache(iQ2), &
3      & K_NS_0(:,:,ixi,4), K_zero(:,:))
```

  - Failure mitigated if `MV_NS(:,:,ixi,iQ2) +` is removed; why?

Interoperability

- There's a mismatch in discretization strategies.
  - PyTorch codebase assumes x and ξ are discretized the same way.
  - Fortran code requires $x \neq \xi$, so grids are misaligned.
  - Need interpolation matrices to wrap Fortran evolution matrices.
- May be technical difficulties deploying Fotran code.
  - I *did* create a Python wrapper via f2py around Fortran code.
  - Compilation requires CMake version ≥ 3.12; not all systems have.
  - Jupyter Notebooks can't locate the compiled `.so` file, no matter what I do to environment variables. (I've been running Fortran code via IPython instead.)

Credits (direct contributions to code/design)

- Daniel Adamiak
- Ian Cloët
- Chris Cocuzza
- Adam Freese
- Nobuo Sato
- Marco Zaccheddu

Thank you for your time!