In simulation process, we solve Poisson boundary problem \( \Delta p = v_{\text{dir}} \) on \( \Omega \in \mathbb{R}^d \), where \( p = g_{\text{dir}} \) on \( \Omega_D \), and \( p = n_h \) on \( \Omega_N \).

Common used methods are Conjugate Gradient (CG) and Preconditioned Conjugate Gradient algorithms.

We purpose is to use Machine Learning to accelerate this process.

**Review of CG**

- CG is an iterative method for approximating the linear system \( Ax = b \).
- Iteratively, it builds \( x_k = x_{k-1} + \alpha_k p_k \), where \( f \) is the objective function
  \[
  f(x) = \frac{1}{2} x^T A x - x^T b
  \]

**Algorithm Conjugate Gradient (Trefethen-Bau)**

1. \( r_0 = b - Ax_0 \), \( p_1 = r_0, k = 1 \)
2. while \( ||r_{k-1}|| \geq \epsilon \) do
3. \( \alpha_k = \frac{r_{k-1}^T r_{k-1}}{p_k^T A p_k} \) (Step Length)
4. \( x_k = x_{k-1} + \alpha_k p_k \) (Solution Update)
5. \( r_k = b - Ax_k \) (Residual Update)
6. \( \beta_k = \frac{r_k^T r_k}{r_{k-1}^T r_{k-1}} \) (Search Direction Correction)
7. \( p_{k+1} = r_k + \beta_k p_k \) (Search Direction Update)
8. \( k = k + 1 \)
9. end while

**Understanding CG**

- The search directions spans the Krylov space \( \{ p_1, p_2, \ldots , p_k \} = \{ b, Ab, A^2 b, \ldots , A^{k-1} b \} = K^k \).
- \( x_k = x_0 + \alpha_1 p_1 + \alpha_2 p_2 + \cdots + \alpha_k p_k \)

- Minimizes \( f(x_k - x^*) \) in \( K^k \).

- **Question:** Can we create better subspace than \( K^k \) using Machine Learning?

**Choice for The Search Direction**

- \( p_k = -\nabla f(x_{k-1}) = b - Ax_{k-1} = r_{k-1} \) is the gradient descent method.
- In CG, \( p_k ') s \) are chosen to be \( A - \text{conjugate} \).
- The 'best' possible choice for \( p_k \) is

\[
    r_k = b - A(x_{k-1} + \alpha_k p_k) = r_{k-1} - \alpha_k A p_k
\]

When \( p_k \) is parallel to \( A^{-1} r_{k-1} \) the algorithm converges in one iteration.

- We propose to pick the search directions using Machine Learning. In other words,

\[
    p_k = ML(r_{k-1})
\]

**Search Direction as ML Output**

- Objective: \( ML(r) \parallel A^{-1} r \).
- Loss \( (r, p) = \| r - A \cdot ML(r) \|_2 \).
- We can directly compute \( \alpha \)

\[
    \argmin_{\alpha} f(A^{-1} r - \alpha p) = \frac{r \cdot p}{p^T A p}
\]

- \( L = \sum_{i \neq D} \| b - r \cdot ML(b) / ML(b)' \cdot A \cdot ML(b)/2 \)

**Implementing ML into Algorithm**

**Algorithm MLALs with A-normalization**

1. \( r_0 = b - Ax_0 \)
2. \( k = 1 \)
3. while \( ||r_{k-1}|| \geq \epsilon \) do
4. \( v = ML(r_{k-1}) \)
5. \( p_k = v - \frac{v^T r_{k-1} + v^T p_{k-1}}{v^T A p_{k-1}} \cdot p_{k-1} - \frac{v^T p_{k-1}}{v^T A p_{k-1}} \cdot p_k \)
6. \( x_k = x_{k-1} + \frac{v^T r_{k-1}}{v^T A p_{k-1}} \cdot p_k \)
7. \( r_k = b - Ax_k \)
8. \( k = k + 1 \)
9. end while

**Dataset Creation**

- The inputs: \( x_1, x_2, \ldots \)
- Dataset is created using approximate eigenvectors of the matrix \( \{ v_1, v_2, \ldots , v_k \} \)

\[
    b_i = c_1 v_1 + c_2 v_2 + \cdots + c_k v_k
\]

Here, \( c_1, c_2, \ldots , c_k \) are picked randomly as follows:

\[
    c_i = \begin{cases} 
    9 \cdot N(0, 1) & \text{if } 1 \leq i \leq \frac{p}{2(1+O(1))} \\
    N(0, 1) & \text{otherwise}
    \end{cases}
\]

and normalized \( b_i = b_i / ||b_i|| \)

**ML Architecture**

- Training is being done with NVIDIA RTX A6000 GPU with 48GB Memory
- Per epoch it takes: 10 minutes for 64^3 grid, 90 minutes for 128^3 grid, 12-16 hours for 256^3 grid.

**Results**

- Architecture for FluidNet

\[
    L = \sum_{i \neq D} \| b - r \cdot ML(b) / ML(b)' \cdot A \cdot ML(b)/2 \]

- \( f_{\text{net}} = \sum_{i} (v \cdot (\frac{1}{2} p_i^2))^{3} \)

- \( \sum_{i=1}^{n} (v \cdot (\frac{1}{2} p_i^2))^{3} \)

- \( f_{\text{net}} = \sum_{i=1}^{n} (v \cdot (\frac{1}{2} p_i^2))^{3} \)

- \( f_{\text{net}} = \sum_{i=1}^{n} (v \cdot (\frac{1}{2} p_i^2))^{3} \)