PETSc and PyTorch interoperability

by Hong Zhang

Integration between simulation and ML

Traditional approaches:

- Save simulation results to files and load them as the training data in a classic ML framework
- Rewrite the simulation codes in a ML framework

These approaches are inefficient and annoying for both computer and human. Can we do in-situ ML for existing simulations?

Instead

We can combine PETSc and PyTorch into a single python based workflow that allows us to

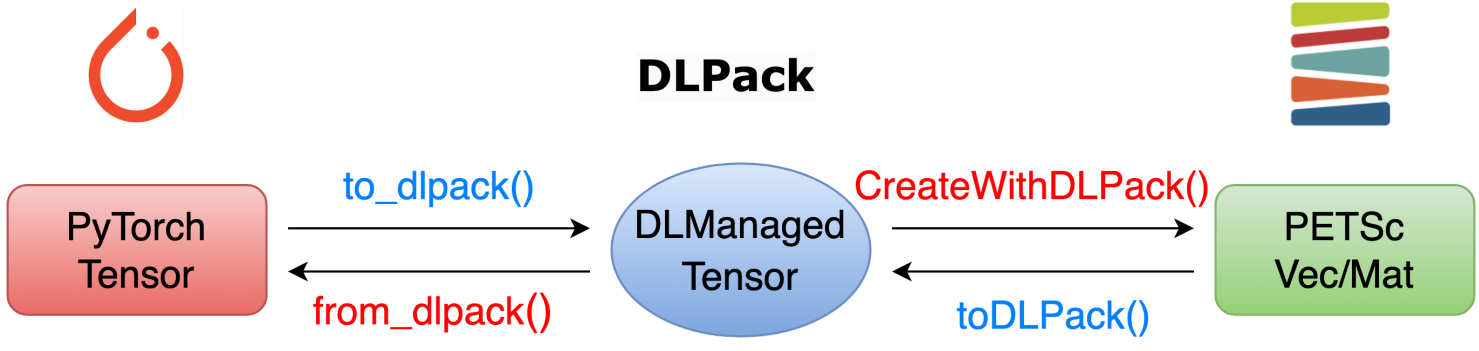
- Embed **machine learning** into **simulation**
- Embed **simulation** into **machine learning**
- Leverage **automatic differentiation** from PyTorch when using PETSc
- Leverage state-of-the-art **numerical solvers** from PETSc when using PyTorch

Topics

- How to convert between PETSc vectors/matrices and PyTorch tensors
- How to generate Jacobian or action of Jacobian with PyTorch and use it in PETSc
- How to use PETSc and PyTorch for solving ODEs and training neural ODEs

Data Structure Conversion Across Libraries

We use the DLPack protocol for efficient conversion between PETSc data structures and tensors.



- Zero-copy in-place conversion
- Pass original objects as arguments directly

Demo: in-place conversion from PyTorch tensor to PETSc Vec

In [2]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
a_tensor = torch.tensor(([1., 2.], [3.,4.]), dtype=torch.float64, devic
a_vec = PETSc.Vec().createWithDLPack(a_tensor) #use a_tensor.detach().c
a_tensor[1][1] = -4.
a_vec.view()
print(a_tensor)
```

```
Vec Object: 1 MPI process
  type: seq
1.
2.
3.
-4.
tensor([[ 1.,  2.],
        [ 3., -4.]], dtype=torch.float64)
```

Demo: in-place conversion from PETSc Vec to PyTorch tensor

In [3]:

```
import torch.utils.dlpack as dlpack
# convert PETSc Vec to tensor
b_vec = PETSc.Vec().createWithArray([5.,6.,7.,8.])
b_vec.attachDLPackInfo(a_vec)
b_tensor = dlpack.from_dlpack(b_vec)
print(b_tensor)
```

```
tensor([[5., 6.],
        [7., 8.]], dtype=torch.float64)
```

Generating the action of Jacobian with PyTorch

- Create the Jacobian matrix with **MatShell**
- Implement your own **MatMult()** operation with autograd from PyTorch
    - Given an input vector X, compute the output vector

$$Y = AX$$

    - Compute the **Jacobian-Vector** product by evaluating the RHS function and then calling *autograd.grad()*
- The Jacobian can be passed to PETSc for solving implicit systems

```
class RHSJacShell:
    def __init__(self, ode):
        self.ode_ = ode

    def mult(self, A, X, Y):
        X.attachDLPackInfo(self.ode_.cached_U)
        x_tensor = dlpack.from_dlpack(X.toDLPack(mode="r"))
        Y.attachDLPackInfo(self.ode_.cached_U)
        y = dlpack.from_dlpack(Y)
        with torch.set_grad_enabled(True):
            self.ode_.cached_u_tensor.requires_grad_(True)
            func_eval = self.ode_.funcEX(self.ode_.t,
self.ode_.cached_u_tensor)
            vjp_u =
torch.autograd.grad(func_eval,self.ode_.cached_u_tensor,x_tensor)
            self.ode_.cached_u_tensor.requires_grad_(False)
        y.copy_(vjp_u[0])
```

```
RHSJac = PETSc.Mat().create()
RHSJac.setSizes([n, n])
RHSJac.setType("python")
shell = RHSJacShell(ode)
RHSJac.setPythonContext(shell)
RHSJac.setUp()
RHSJac.assemble()
ode.ts.setRHSJacobian(ode.evalRHSJacobian, RHSJac)
```

Solving a system of ODEs with PETSc and PyTorch

As an example, we solve Robertson's equations with PNODE (a neural ODE solver based on PETSc and PyTorch).

$$\begin{aligned} \frac{du_1}{dt} &= -k_1 u_1 + k_3 u_2 u_3, \\ \frac{du_2}{dt} &= k_1 u_1 - k_2 u_2^2 - k_3 u_2 u_3, \\ \frac{du_3}{dt} &= k_2 u_2^2, \end{aligned} \tag{1}$$

where u are concentrations and k are reaction rate constants.

In the implementation, the RHS function inherits from **nn.module** so it can be backpropgated to generate the Jacobian. System parameters are stored in a differentiable tensor. A matrix-free Jacobian is computed automatically with PyTorch.

In [5]:

```
class RHSFunc(nn.Module):
    def __init__(self):
        super(RHSFunc, self).__init__()
        self.k = nn.Parameter(
            torch.tensor([0.05, 4e7, 2e4], requires_grad=True, dtype=tc
        )

    def forward(self, t, u):
        k1 = self.k[0]
        k2 = self.k[1]
        k3 = self.k[2]
        f1 = -k1 * u[0] + k3 * u[1] * u[2]
        f2 = k1 * u[0] - k3 * u[1] * u[2] - k2 * u[1] ** 2
        f3 = k2 * u[1] ** 2
        return torch.stack((f1, f2, f3), -1)
```

Sensivitiy analysis for the ODE system

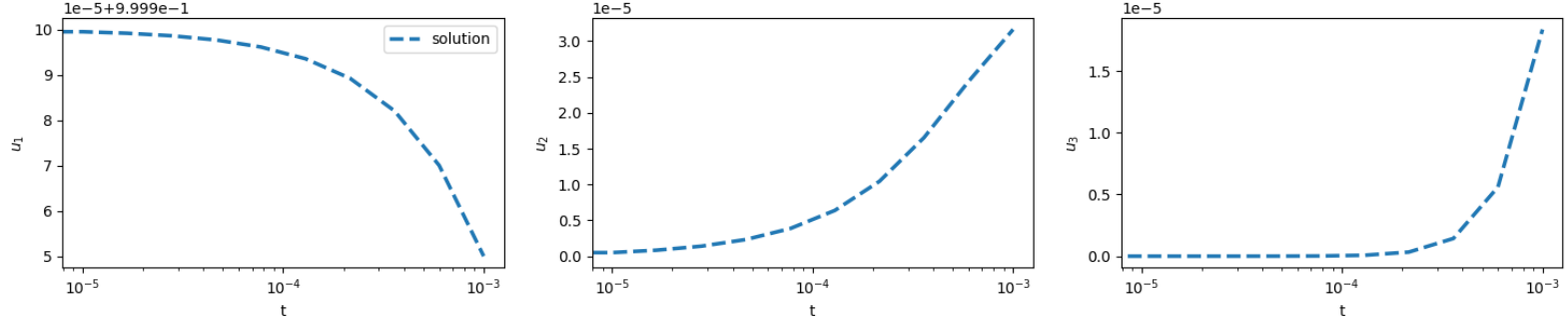
With PETSc TSAdjoint solvers we can calculate the **derivatives** of a quantity of interest(e.g. a loss function) with respect to the system parameters. PNODE encapsulates PETSc TS solvers and TSAdjoint solvers. It requires users to

- Provide the RHS function for the ODE
- Store the system parameters in a tensor with *requires_grad=True*
- Define the quantity of interest

The adjoint methods allow us to **avoid backpropgating the entire ODE solver** while **maintaining reverse-accuracy and memory-efficiency**.

In [7]:
```
initial_state = torch.tensor([1.0, 0.0, 0.0], dtype=torch.float64, devi
func = RHSFunc().to(device)
solver = petsc_adjoint.ODEPetsc()
solver.setupTS(initial_state, func, step_size=step_size, method="cn", e
u = solver.odeint_adjoint(initial_state, t)
visualize(t, u.detach(), "solution")
loss = torch.mean(torch.abs(u - true_u))
loss.backward()
for p in func.parameters(): print(p.grad)
```

```
tensor([ 1.4257e-04, -1.4641e-18, -4.1032e-15], dtype=torch.floa
t64)
```



Training neural ODEs

Neural ODE is a methodology that fuses differential equations and neural networks. In this approach, the RHS $f(t, x, p)$ is approximated by a neural network whose parameters can be trained.

Assuming the RHS of the Robertson's equations is unknown, we approximate it using a NN with 5 hidden layers with 5 nodes per layer and an activation function of GELU.

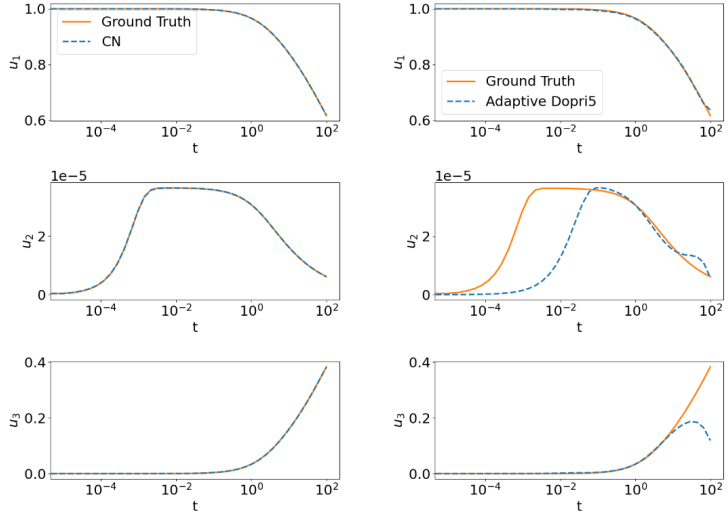
```
class ODEFunc(nn.Module):
    def __init__(self):
        super(ODEFunc, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(3, 5, bias=False), nn.GELU(),
            nn.Linear(5, 5, bias=False), nn.GELU(),
            nn.Linear(5, 5, bias=False), nn.GELU(),
            nn.Linear(5, 5, bias=False), nn.GELU(),
            nn.Linear(5, 5, bias=False), nn.GELU(),
            nn.Linear(5, 5, bias=False), nn.GELU(),
            nn.Linear(5, 3, bias=False),)

    def forward(self, t, u):
        return self.net(u)
```

After setting up the neural ODE model and the optimizer, we start the optimization loop for training.

```
func_PNODE = ODEFunc().to(device)
ode_PNODE = petsc_adjoint.ODEPetsc()
ode_PNODE.setupTS(y0, func_PNODE, step_size=step_size, method="cn",
enable_adjoint=True, implicit_form=True)
optimizer_PNODE = optim.AdamW(func_PNODE.parameters(), lr=5e-3)
for itr in range(curr_iter, niters + 1):
    batch_t, batch_u = get_batch()
    optimizer_PNODE.zero_grad()
    pred_u_PNODE = ode_PNODE.odeint_adjoint(u0, batch_t)
    loss_PNODE = torch.mean(torch.abs(pred_u_PNODE - batch_u))
    loss_PNODE.backward()
    optimizer_PNODE.step()
```

Results



(a) Implicit method

(b) Explicit method

- This is a stiff dynamical sytem for which training with implicit methods work better than training with explicit methods
- Using adjoint-capable PETSc time integrators is the key to achiving performance
- One can fine-tune linear solvers, nonlinear solvers, and checkpointing for adjoint calculation with **command-line options**
- PETSc and PyTorch work together with minimal conversion overhead, full **GPU** support and **mini-batching** support

Sources for more information

- https://github.com/caidao22/pnode
- H. Zhang and W. Zhao. *A Memory-Efficient Neural Ordinary Differential Equation Framework Based on High-Level Adjoint Differentiation*. IEEE Transactions on Artificial Intelligence, pp. 1–11, 2022.
- H. Zhang, E. M. Constantinescu, and B. F. Smith. *PETSc TSAdjoint: A Discrete Adjoint ODE Solver for First-Order and Second-Order Sensitivity Analysis*. SIAM Journal on Scientific Computing. 44.1, pp. C1–C24, 2022
- H. Zhang and E. M. Constantinescu. *Optimal checkpointing for adjoint multistage time-stepping schemes*. Journal of Computational Science. 66, p. 101913, 2023.