Time Integration in BOUT++

Nick Walkden on behalf of Ben Dudson and the BOUT++ Team

Contents

- Numerical Time Integration
- Time solvers in BOUT++
- Optimization: Physics based preconditioning
- Summary

Numerical Time Integration

- A general evolution equation of the form

$$\partial_t \boldsymbol{f} = \boldsymbol{F}(\boldsymbol{f}) \qquad f = \begin{pmatrix} n \\ v \\ etc \end{pmatrix}$$

can be solved numerically by discretizing the time domain

- This can be done equivalently by

Explicit methods

$$\partial_t \boldsymbol{f} \approx \frac{\boldsymbol{f}^{n+1} - \boldsymbol{f}^{n}}{\Delta t} \approx \boldsymbol{F}(\boldsymbol{f}^{n})$$

Implicit methods

$$\partial_t \boldsymbol{f} \approx \frac{\boldsymbol{f}^{n} - \boldsymbol{f}^{n-1}}{\Delta t} \approx \boldsymbol{F}(\boldsymbol{f}^{n})$$

or by combining the two using

ImEx methods

$$\partial_t \boldsymbol{f} = \boldsymbol{F}_{\boldsymbol{slow}}(\boldsymbol{f}) + \boldsymbol{G}_{\boldsymbol{fast}}(\boldsymbol{f}) = \boldsymbol{F}_{\boldsymbol{exp}}(\boldsymbol{f}) + \boldsymbol{G}_{\boldsymbol{imp}}(\boldsymbol{f})$$

Numerical Time Integration

Explicit Methods

$$\partial_t \boldsymbol{f} \approx \frac{\boldsymbol{f}^{n+1} - \boldsymbol{f}^{n}}{\Delta t} \approx \boldsymbol{F}(\boldsymbol{f}^{n})$$

Good for:

- Single timescales
- Non-linear problems

Bad for:

- Stiff problems

Advantages:

- Easy to implement
- Relatively little cost per timestep

Disadvantages:

- Stability (ie CFL condition)

Numerical Time Integration

Implicit Methods

$$\partial_t \boldsymbol{f} \approx \frac{\boldsymbol{f}^n - \boldsymbol{f}^{n-1}}{\Delta t} \approx \boldsymbol{F}(\boldsymbol{f}^n)$$

Good for:

- Stiff problems/multiple timescales

Bad for:

- Heavily non-linear problems

Advantages:

- Unconditionally stable
- Large timesteps

Disadvantages:

- Complexity
- Cost per timestep
- Under-relaxation

Numerical Time Integration

ImEx Methods

$$\partial_t f = F_{exp}(f) + G_{imp}(f)$$

Good for:

- Stiff problems/multiple timescales

Bad for:

Advantages:

- Timestep only limited by slow scales

Disadvantages:

- Complexity
- Cost per timestep
- Dependant on physics model

Numerical Time Integration

- Implicit Methods are often cast as a Newton iteration

$$\boldsymbol{G}(\boldsymbol{f}^n) = \boldsymbol{f}^n - \boldsymbol{f}^{n-1} - \gamma \boldsymbol{F}(\boldsymbol{f}^n) = 0$$

can be Taylor expanded to give

$$\boldsymbol{G}(\boldsymbol{f}^n) \approx \boldsymbol{G}(\boldsymbol{f}^n{}_m) + (\boldsymbol{f}^n - \boldsymbol{f}^n{}_m)\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{f}^n} = 0$$

where

$$\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{f}^n} = I - \gamma J \qquad J = \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{f}^n} = \begin{pmatrix} \frac{\partial F_1}{\partial f_1} & \cdots & \frac{\partial F_N}{\partial f_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_1}{\partial f_N} & \cdots & \frac{\partial F_N}{\partial f_N} \end{pmatrix}$$

Numerical Time Integration

- Implicit Methods are often cast as a Newton iteration

  So we can now solve the problem

$$(\boldsymbol{I} - \gamma \boldsymbol{J})\boldsymbol{\delta f}^n = -\boldsymbol{G}(\boldsymbol{f}^n)$$

  which we use to update

$$\boldsymbol{f}^n{}_{m+1} = \boldsymbol{f}^n{}_m + \boldsymbol{\delta f}^n$$

  iteratively until

$$\boldsymbol{G}(\boldsymbol{f}^n) = 0$$

Time Solvers in BOUT++

- To advance in time, time derivatives of evolving quantities are required
  - All fields store the derivatives in another field called `deriv` which can be accessed through `var.timeDeriv()`

```
Field3D var;
Field3D *deriv = var.timeDeriv()
```

  - When we call ddt(var) we are actually calling the inline function (include/field3d.hxx line 346)

```
inline Field3D& ddt(Field3D &f){
   return *(f.timeDeriv()); }
```

Time Solvers in BOUT++

- To advance in time, time derivatives of evolving quantities are required
  - All fields store the derivatives in another field called `deriv` which can be accessed through `var.timeDeriv()`

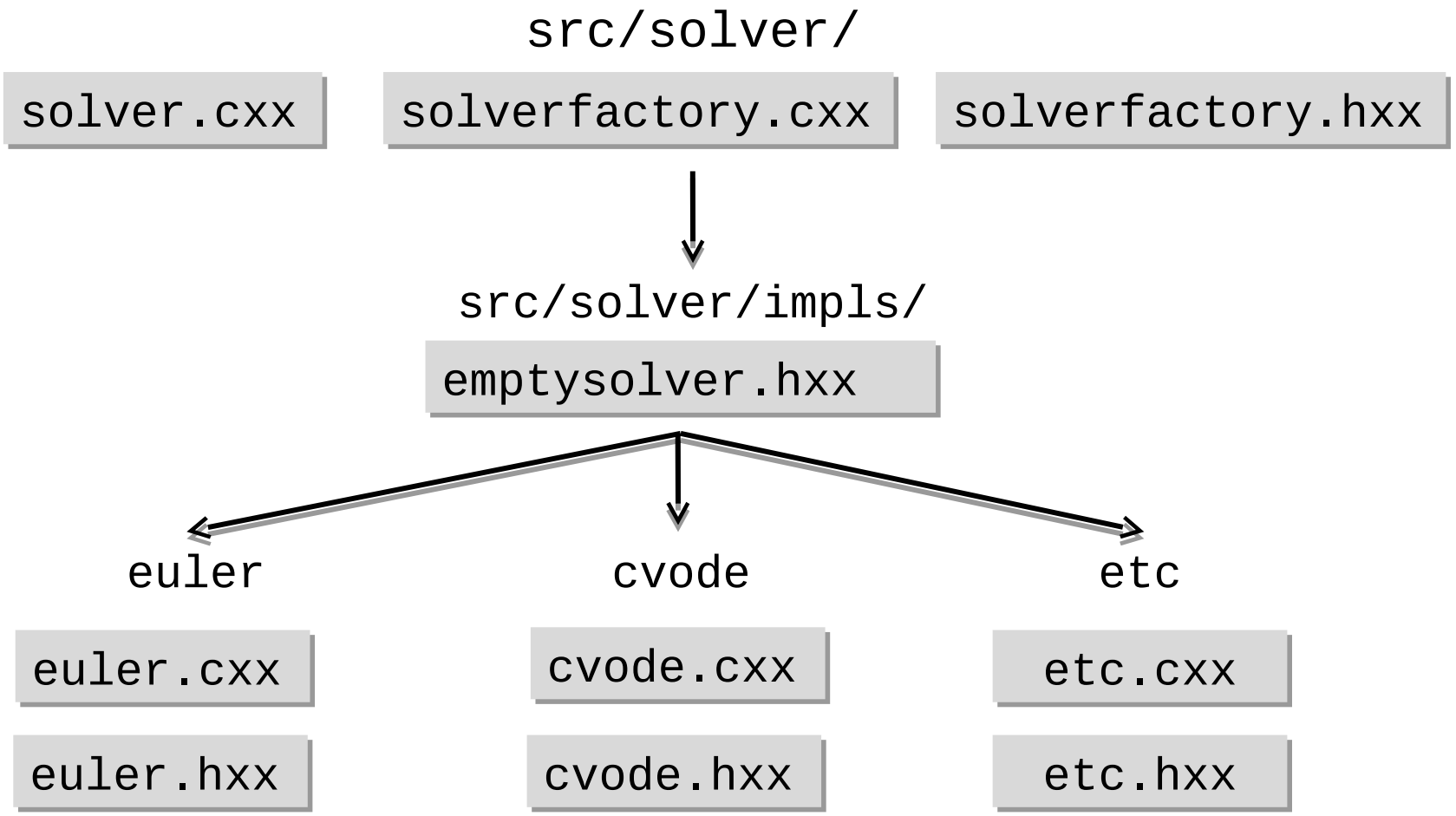
This allows us to treat `ddt(var)` as a variable in our code so that we can write statements like

```
ddt(var) = … + …
```

```
inline Field3D& ddt(Field3D &f){
   return *(f.timeDeriv()); }
```

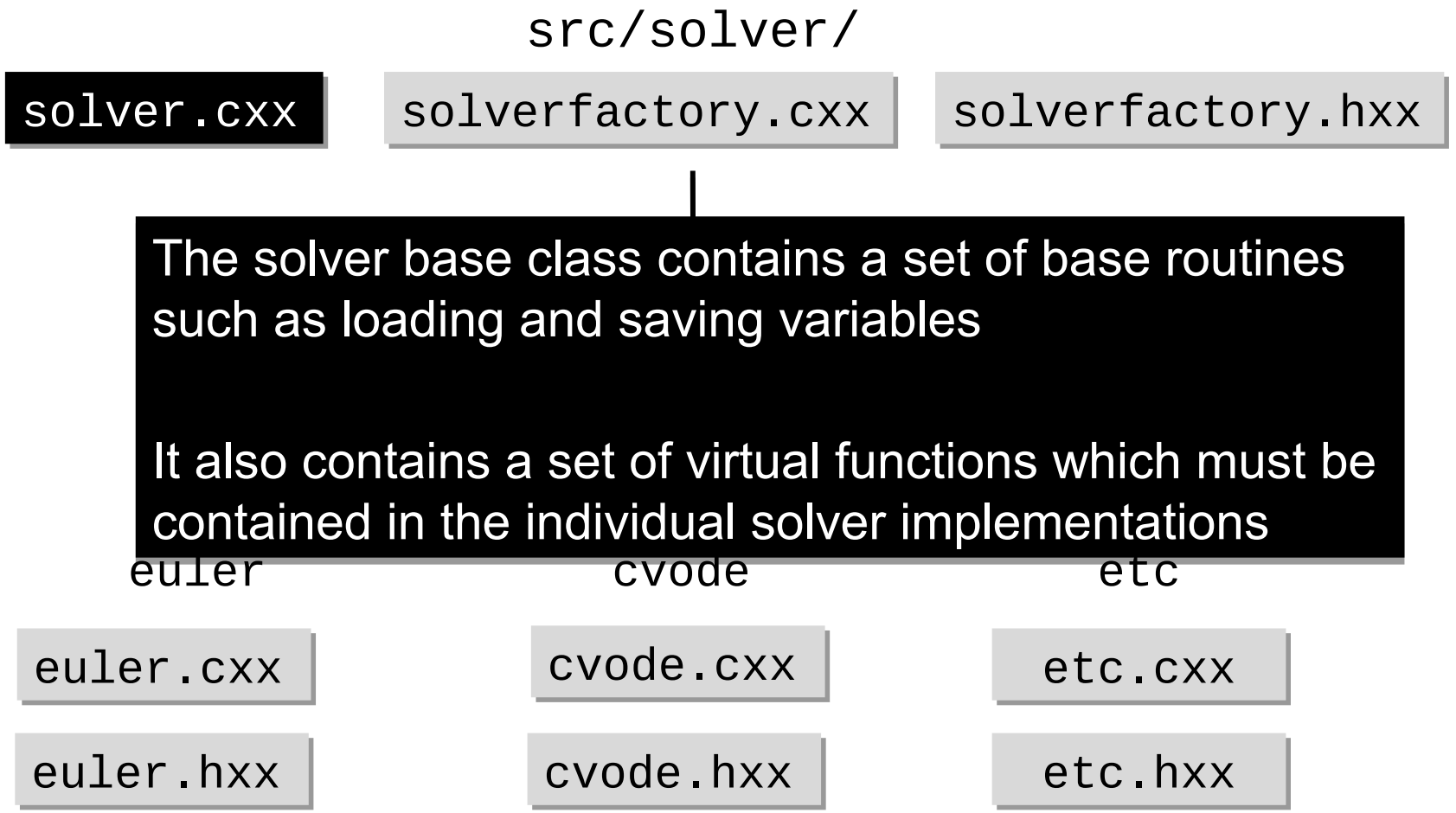
Time Solvers in BOUT++

- In BOUT++ the solver has a factory format



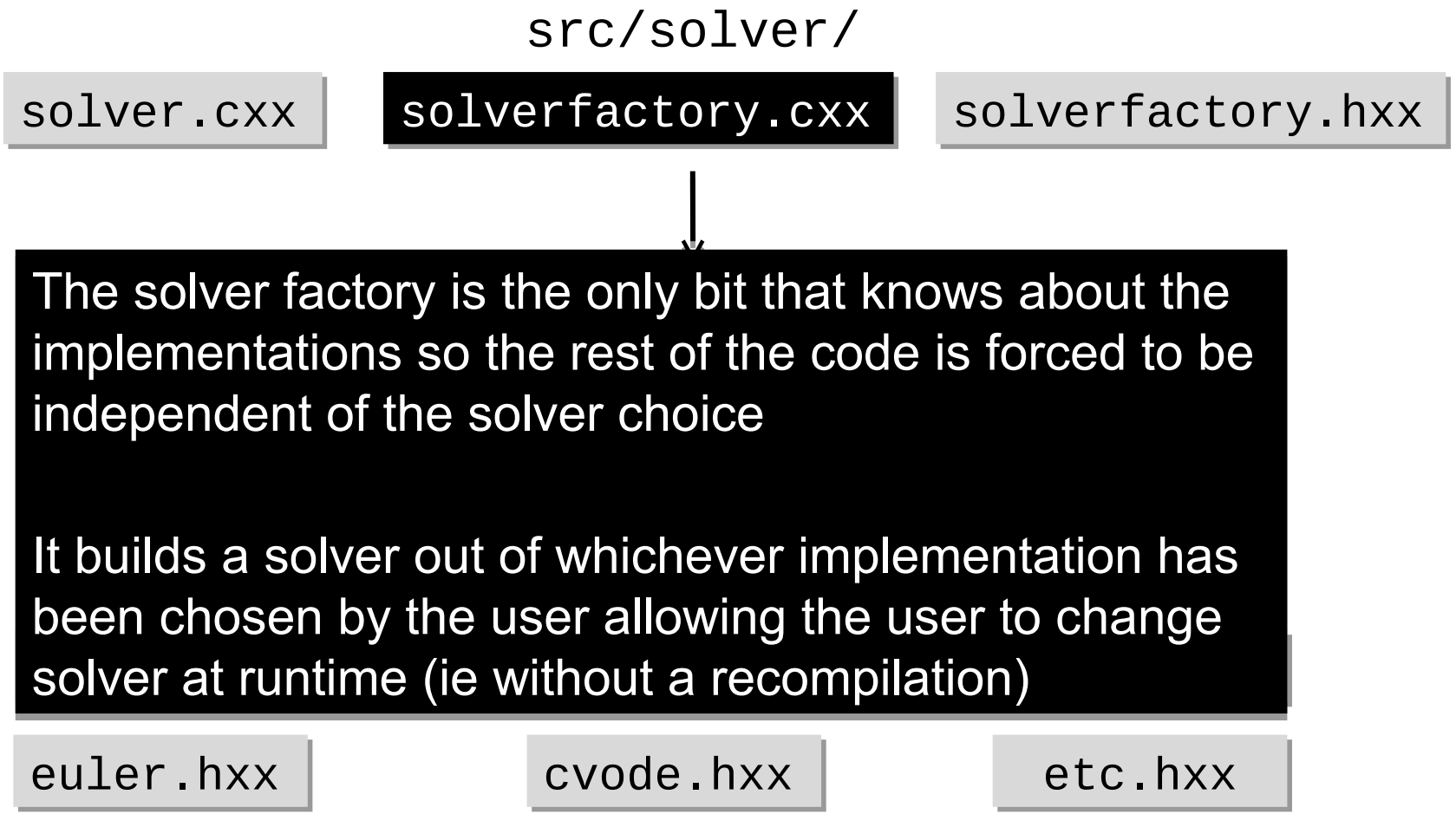
Time Solvers in BOUT++

- In BOUT++ the solver has a factory format



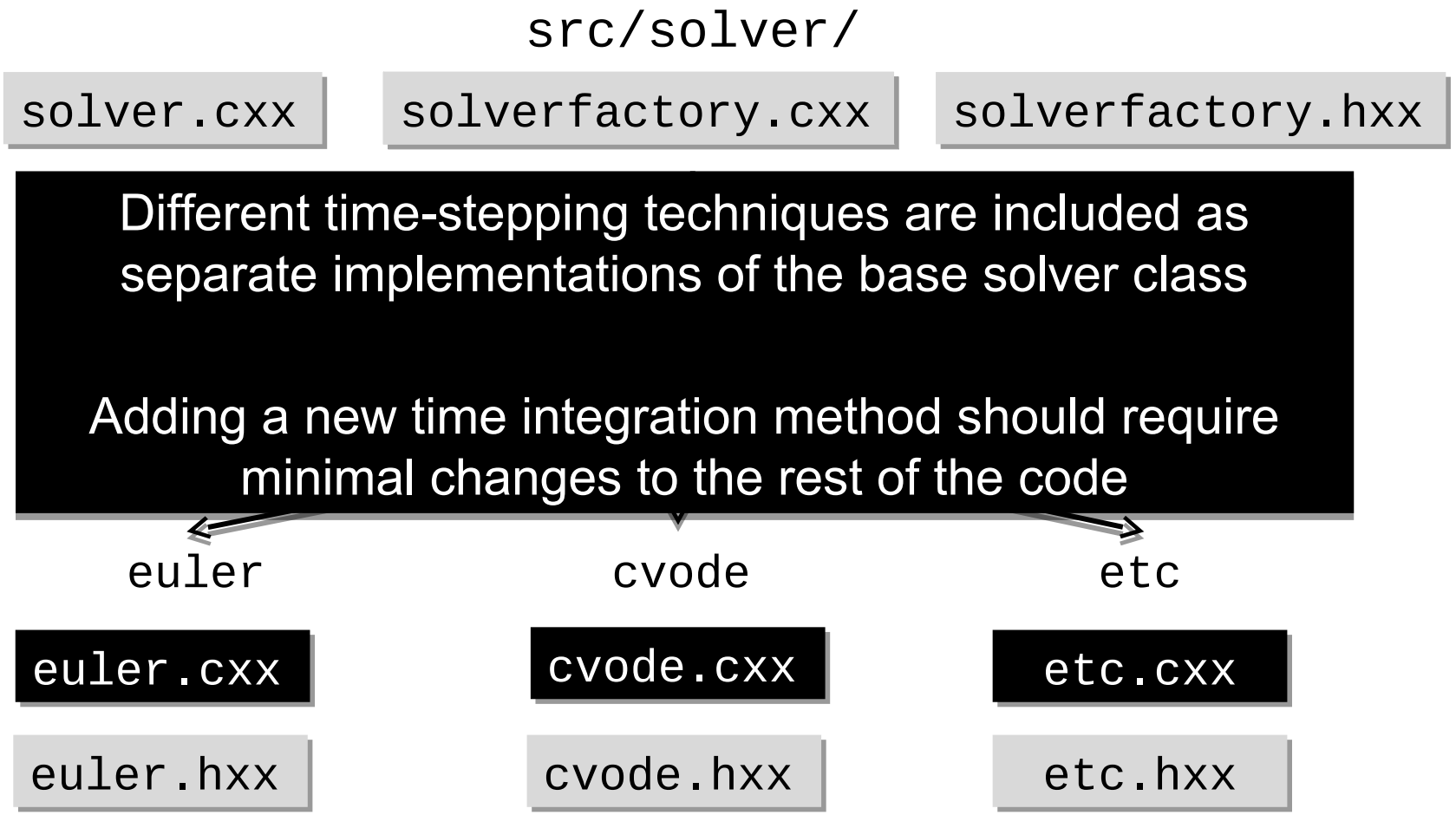
Time Solvers in BOUT++

- In BOUT++ the solver has a factory format



Time Solvers in BOUT++

- In BOUT++ the solver has a factory format



Time Solvers in BOUT++

- BOUT++ has a range of solver implementations:

Explicit Solvers	Implicit Solvers	ImEx Solvers
`euler`	`pvode`	`arkode`
`rk4`	`cvode`	`BDF multistep`
`rk3-ssp`	`petsc (various)`	
`karniadakis`	`ida`	
`power`		

The default behaviour is to use either CVODE or IDA if present, otherwise use PVODE

Time Solvers in BOUT++

- BOUT++ has a range of solver implementations:

Explicit Solvers	Implicit Solvers	ImEx Solvers
euler	pvode	arkode
rk4	cvode	BDF multistep
rk3-ssp	petsc (various)	
karniadakis	ida	
power		

Require external libraries

The default behaviour is to use either CVODE or IDA if present, otherwise use PVODE

Time Solvers in BOUT++

- BOUT++ has a range of solver implementations:
- The solver type is set in the `BOUT.inp` file or on the command line

In BOUT.inp:

```
[solver]
type = …
```

On the command line

```
./executable
solver:type=…
```

- Usually best to try a few (ie RK4 and pvode) to find the optimal choice for your case

Time Solvers in BOUT++

- Some additional useful options are:

Option	Function
`mxstep = 500`	Number of timesteps to try before timestep is a failure
`atol = 1e-10`	Absolute tolerance used to determine error norm. Determines noise level of solution
`rtol = 1e-5`	Relative tolerances used to determine error norm. Indicates no of digits of relative accuracy for a single time step
`adaptive = false`	Use adaptive timestepping in rk4
`use_precon = true`	Use preconditioning in cvode and petsc

Time Solvers in BOUT++

- To tell the time-solver to solve for a particular field we use the routine

```
bout_solve(n, ”density”);
```

or the macro `SOLVE_FOR(n);`

or similarly for 2 <= n <= 6 `SOLVE_FORn(n,…);`

- This calls a routine in the solver class

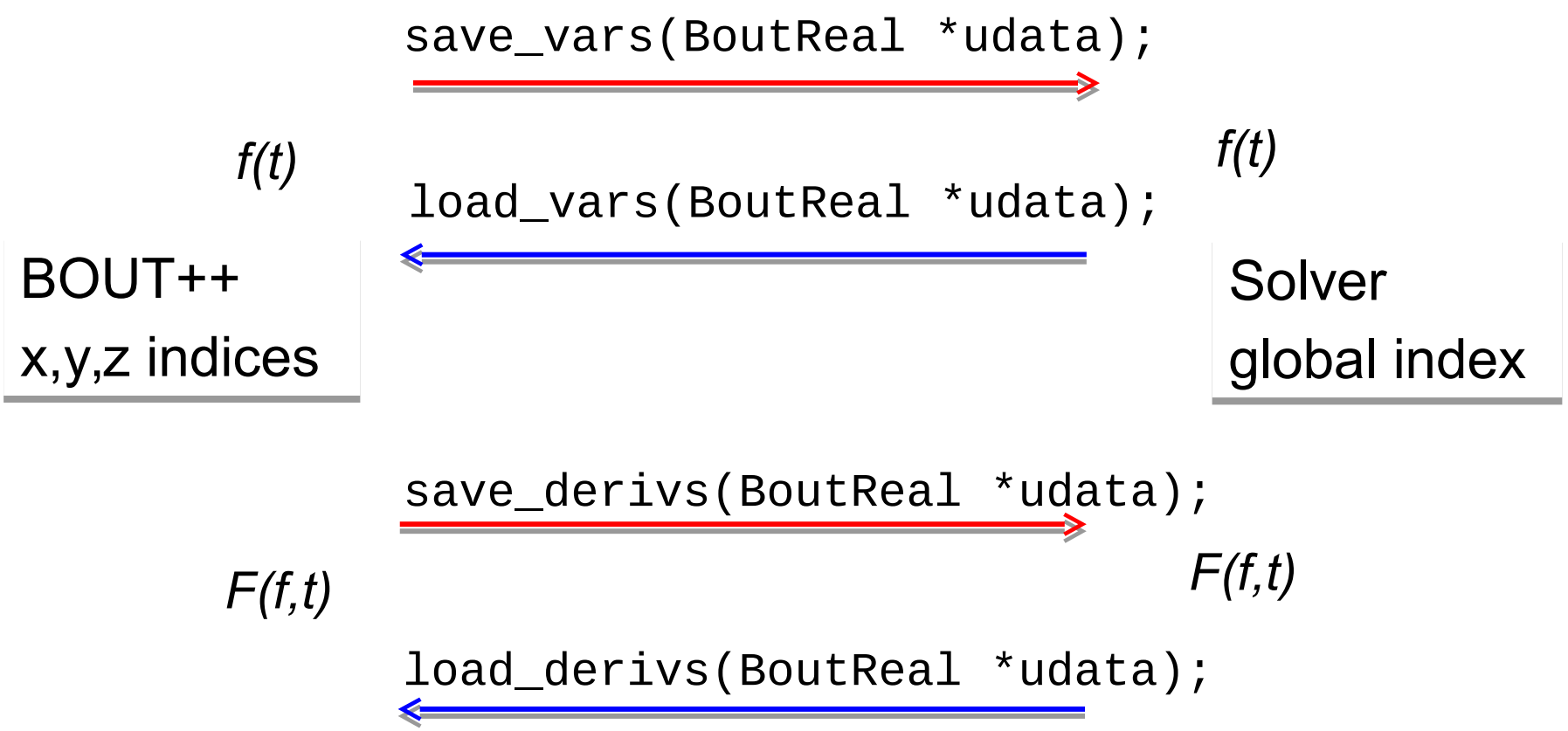
```
solver->add(n,”density”);
```

which adds the variable to the state and residual vectors for

input to the time-solver

Time Solvers in BOUT++

- BOUT++ data is then be passed to the solver through a few protected functions



Time Solvers in BOUT++

- There are two mandatory functions that a time-solver implementation must contain

```
int mysolver::init(bool restarting, int nout, BoutReal
                   tstep)
```

Initialization of the solver. Calls a generic solver initialization as well as implementation specific solver options.

Time Solvers in BOUT++

- There are two mandatory functions that a time-solver implementation must contain

```
int mysolver::init(bool restarting, int nout, BoutReal
                   tstep)
int mysolver::run()
```

Running the solver. This function integrates in time until nout is reached and the simulation is over.

Time Solvers in BOUT++

- There are two mandatory functions that a time-solver implementation must contain
- A call to `int bout_run(Solver *solver, rhsfunc physics_run)` (`bout++.cxx`, L 287) is made in the main function which runs the solver
- That's how the magic happens…

Time Solvers in BOUT++

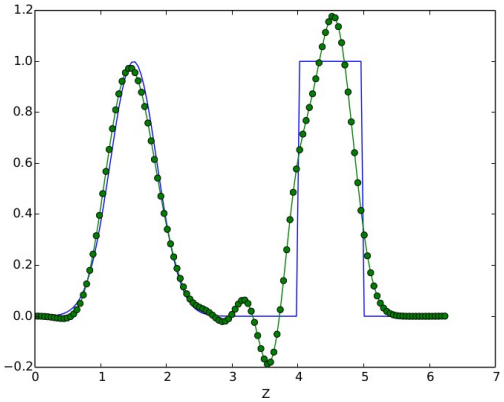
- For many problems the implicit PVODE/CVODE solvers work well
- In some cases however they can fail

Example: Advection of a pulse in 1D $\frac{\partial f}{\partial t} = -v \frac{\partial f}{\partial x}$

PVODE integrator, absolute tolerance 1e-12, relative tolerance 1e-5

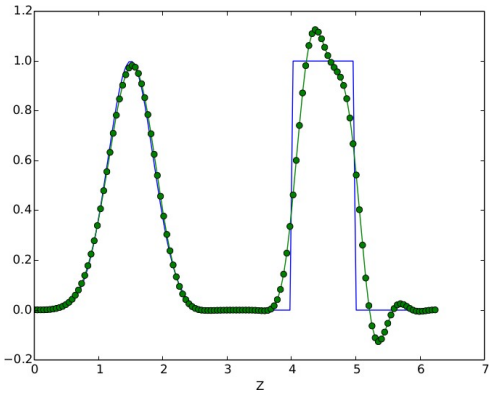
CFL condition limits explicit methods to 128 steps, or 512 evaluations for RK4
The RK3-SSP method requires dt < 0.2 dt(CFL), or 1920 evaluations

PVODE: 2904 evals



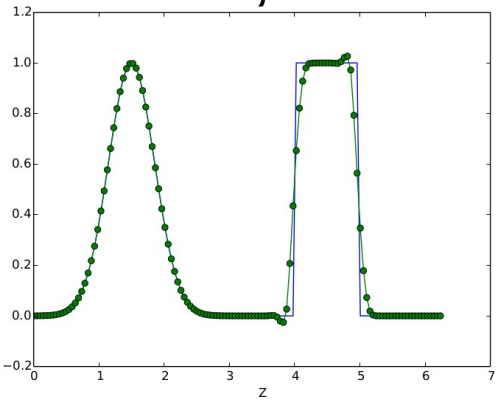
Central differencing with dissipation

PVODE: 3073 evals



Fromm (2nd order)

PVODE: **75,585** evals



XPPM (nonlinear limiter)

Time Solvers in BOUT++

- For ImEx schemes the split operators must be defined

C style interface

```
examples/split-operator/

int physics_init(bool restart) {
    Solver->setSplitOperator
        (physics_run, reaction);
    ...
}

int physics_run(BoutReal time) {
    // Explicit part
}

int reaction(BoutReal time) {
   // Implicit part
}
```

C++ style interface

```
examples/test-drift/

class DriftWave : public
PhysicsModel {
    int init(bool restart) {
        SetSplitOperator();
        ...
    }
    int convective(BoutReal time) {
        // Explicit part
    }
    int diffusive(BoutReal time) {
        // Implicit part
    }
}
```

Optimization

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%." D. Knuth

- Optimization in BOUT++ is handled almost entirely internally and should rarely be tinkered with

- One exception to this is preconditioning

Optimization

- Recall that the implicit time-solver(s) solves a newton iteration of the form

$$(I - \gamma J)\boldsymbol{\delta f} = -\boldsymbol{G}(f)$$

which requires the inversion of $(I - \gamma J)$

- It may be possible to define a new operator, *P*, such that

$$P(I - \gamma J)\boldsymbol{\delta f} = -P\boldsymbol{G}(f)$$

If $P(I - \gamma J)$ is easier to invert then we may speed up our solver!!

Optimization

- We can use our knowledge of the physics system to help us here

- If we have some stiff physics on some timescale then we can try and construct *P* such that

$$P \approx (I - \gamma J)^{-1}$$

over

- This then means that over the time-scale τ the inversion is trivial

$$P(I - \gamma J)\boldsymbol{\delta f} = -P\boldsymbol{G}(f) \quad \xrightarrow{\tau} \quad I\boldsymbol{\delta f} = -(I - \gamma J)^{-1}\boldsymbol{G}(f)$$

Optimization

- Problem: How do we find P ?
  1. Reduce equations
  2. Calculate analytical Jacobian matrix
  3. Factorize matrix using schur factorization
  4. Simplify the problem (decouple perpendicular and parallel dynamics)

Optimization

1. Reduce equations

- $\partial_t n = f(n,T)$

$$\partial_t T = g(n,T) + \partial_{||}\chi_{||}\partial_{||}T$$

$$\chi_{||} = \chi_0 T^{5/2}$$

Slow (non-stiff) physics

Fast (stiff) physics

We want to isolate the fast physics, so consider the reduced system

- $\partial_t n = 0$

$$\partial_t T = \partial_{||}\chi_{||}\partial_{||}T$$

Optimization

2. Calculate the Jacobian

$$J = \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{f}} = \begin{pmatrix} \partial_n \partial_t n & \partial_n \partial_t T \\ \partial_T \partial_t n & \partial_T \partial_t T \end{pmatrix}$$

For our reduced system

$$J = \begin{pmatrix} 0 & 0 \\ 0 & \chi_0 T^{5/2} \partial_{||}{}^2 \end{pmatrix}$$

So

$$I - \gamma J = \begin{pmatrix} 1 & 0 \\ 0 & 1 - \gamma \chi_0 T^{5/2} \partial_{||}{}^2 \end{pmatrix}$$

Optimization

3. Factorize the matrix

Using a technique called Schur factorization

$$\begin{pmatrix} E & U \\ L & D \end{pmatrix}^{-1} = \begin{pmatrix} I & -E^{-1}U \\ 0 & I \end{pmatrix} \begin{pmatrix} E^{-1} & 0 \\ 0 & S^{-1} \end{pmatrix} \begin{pmatrix} I & 0 \\ -LE^{-1} & I \end{pmatrix}$$

Where the Schur complement is

$$S = D - LE^{-1}U$$

For our case $E = 1, L = U = 0$ and $\mathrm{D} = 1 - \gamma\chi_0 T^{\frac{5}{2}} \partial_{||}{}^2$ so

$$S^{-1} = \left(1 - \gamma\chi_0 T^{\frac{5}{2}} \partial_{||}{}^2\right)^{-1}$$

Optimization

- Problem: How do we find *P* ?
  1. Reduce equations
  2. Calculate analytical Jacobian matrix
  3. Factorize matrix using schur factorization
  4. Simplify the problem (decouple perpendicular and parallel dynamics)

We now have the preconditioner operator *P*

- $P = \begin{pmatrix} 1 & 0 \\ 0 & S^{-1} \end{pmatrix}$ where where
- $S^{-1} = \left(1 - \gamma \chi_0 T^{\frac{5}{2}} \partial_{||}{}^2\right)^{-1}$

Optimization

- Problem: How do we apply the preconditioner?
  5. Create `precon` function
  6. Implement preconditioning operator on time derivatives
  7. Tell the solver to use a preconditioner
  8. Tinker with the preconditioner until you get some speedup

Optimization

5. Write a precon function

At the end of your physics module you now need a function called

```
int precon(BoutReal t, BoutReal gamma, BoutReal delta)
```

This function must apply the preconditioning operator to the time derivatives of the evolving variables

`t` is the current simulation time, `gamma` is the methods timestep and `delta` may be used to apply constraints (but rarely used, so don't worry about it)

Optimization

6. Apply preconditioner to time derivatives

To apply the preconditioner we will need to invert a matrix in the parallel direction. BOUT++ has a class to do this, called `InvertPar` which solves

```
InvertPar *precon_inv;
int physics_init(bool restarting){
…
precon_inv = InvertPar::create();
precon_inv->setCoefA(1.0);
…
}
```

Optimization

6. Apply preconditioner to time derivatives

Now in the `precon` function we apply the operator

```
int precon(BoutReal t, BoutReal gamma, BoutReal
delta){

mesh->communicate(ddt(T));
Field 2D B;
B = -gamma*chi*(T.DC())^(5./2.);
precon_inv->setCoefB(B);
ddt(T) = precon_inv->solve(ddt(T));
ddt(T).applyBoundary("neumann_o2");

}
```

Optimization

7. Tell the solver to use the preconditioner

In `BOUT.inp` we need the lines:

```
[solver]
type = cvode #or petsc
use_precon = true
rightprec = false
```

In `physics_init` we need the line:

```
solver->setPrecon(precon);
```

Optimization

7. Tell the solver to use the preconditioner

In a 1D high powered SOL slab equilibrium calculation on 8 cores

Solver	Setup	Wall time (s)	~Iteration count
CVODE	Isothermal	2	100
CVODE	Conduction removed	5	300
PVODE	None	221	37000
CVODE	None	276	45000
CVODE	BBD Preconditioner	58	17000
CVODE	Custom Preconditioner	9	800

Approximately 30x speedup with preconditioner

Optimization

8. Tinker

Since the preconditioner affects convergence, but **not** the solution, you can tinker with it to find the best setup

WARNING: Because the preconditioner is problem dependant it will not always be beneficial

Summary

- BOUT++ contains a suite of time-solvers including explicit and implicit options
- The choice of solver is problem dependant but can be interchanged at runtime
- Physics based preconditioning can be used to optimize (implicit) solvers, but costs (some) blood