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
  \[ \partial_t f = F(f) \]
  where
  \[ f = \begin{pmatrix} n \\ v \\ \text{etc} \end{pmatrix} \]
  can be solved numerically by discretizing the time domain.

- This can be done equivalently by:
  - **Explicit methods**
    \[ \partial_t f \approx \frac{f^{n+1} - f^n}{\Delta t} \approx F(f^n) \]
  - **Implicit methods**
    \[ \partial_t f \approx \frac{f^n - f^{n-1}}{\Delta t} \approx F(f^n) \]
  or by combining the two using:

- **ImEx methods**
  \[ \partial_t f = F_{\text{slow}}(f) + G_{\text{fast}}(f) = F_{\text{exp}}(f) + G_{\text{imp}}(f) \]
Explicit Methods

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)

\[ \frac{\partial f}{\partial t} \approx \frac{f^{n+1} - f^n}{\Delta t} \approx F(f^n) \]
**Implicit Methods**

Good for:
- Stiff problems/multiple timescales

Bad for:
- Heavily non-linear problems

Advantages:
- Unconditionally stable
- Large timesteps

Disadvantages:
- Complexity
- Cost per timestep
- Under-relaxation

\[
\frac{\partial_t f}{\Delta t} \approx \frac{f^n - f^{n-1}}{\Delta t} \approx F(f^n)
\]
**ImEx Methods**

**Good for:**
- Stiff problems/multiple timescales

**Bad for:**

**Advantages:**
- Timestep only limited by slow scales

**Disadvantages:**
- Complexity
- Cost per timestep
- Dependant on physics model

\[
\dot{f} = F_{exp}(f) + G_{imp}(f)
\]
• Implicit Methods are often cast as a Newton iteration

\[ g(f^n) = f^n - f^{n-1} - \gamma F(f^n) = 0 \]

can be Taylor expanded to give

\[ g(f^n) \approx g(f^n_m) + (f^n - f^n_m) \frac{\partial g}{\partial f^n} = 0 \]

where

\[ \frac{\partial g}{\partial f^n} = I - \gamma J \]

\[ J = \frac{\partial F}{\partial f^n} = \begin{pmatrix} \frac{\partial F_1}{\partial f_1} & \cdots & \frac{\partial F_N}{\partial f_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_1}{\partial f_N} & \cdots & \frac{\partial F_N}{\partial f_N} \end{pmatrix} \]
Implicit Methods are often cast as a Newton iteration

So we can now solve the problem

\((I - \gamma J) \delta f^n = -G(f^n)\)

which we use to update

\(f_{n+1}^m = f_n^m + \delta f^n\)

iteratively until

\(A(f^n) = 0\)
• 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());
    }
• 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

\[
\text{ddt(var)} = \ldots + \ldots
\]

inline Field3D& ddt(Field3D &f){
    return *(f.timeDeriv());
}
• In BOUT++ the solver has a factory format

```
src/solver/
  solver.cxx  solverfactory.cxx  solverfactory.hxx
```

```
src/solver/impls/
  emptysolver.hxx
```

```
euler  cvode  etc
  euler.cxx  cvode.cxx  etc.cxx
  euler.hxx  cvode.hxx  etc.hxx
```
In BOUT++ the solver has a factory format

src/solver/

- solver.cxx
- solverfactory.cxx
- solverfactory.hxx

The solver base class contains a set of base routines such as loading and saving variables.

It also contains a set of virtual functions which must be contained in the individual solver implementations.

euler
euler.cxx
euler.hxx
cvode
cvode.cxx
cvode.hxx
etc
etc.cxx
etc.hxx
• In BOUT++ the solver has a factory format

```
src/solver/
solver.cxx  solverfactory.cxx  solverfactory.hxx
```

The solver factory is the only bit that knows about the implementations so the rest of the code is forced to be independent of the solver choice.

It builds a solver out of whichever implementation has been chosen by the user allowing the user to change solver at runtime (ie without a recompilation)
• In BOUT++ the solver has a factory format

```
src/solver/
solver.cxx  solverfactory.cxx  solverfactory.hxx
```

Different time-stepping techniques are included as separate implementations of the base solver class

Adding a new time integration method should require minimal changes to the rest of the code

```
euler
euler.cxx  euler.hxx

cvode
cvode.cxx  cvode.hxx

etc
etc.cxx  etc.hxx
```
**Time Solvers in BOUT++**

- BOUT++ has a range of solver implementations:

<table>
<thead>
<tr>
<th>Explicit Solvers</th>
<th>Implicit Solvers</th>
<th>ImEx Solvers</th>
</tr>
</thead>
<tbody>
<tr>
<td>euler</td>
<td>pvode</td>
<td>arkode</td>
</tr>
<tr>
<td>rk4</td>
<td>cvode</td>
<td>BDF multistep</td>
</tr>
<tr>
<td>rk3-ssp</td>
<td>petsc (various)</td>
<td></td>
</tr>
<tr>
<td>karniadakis</td>
<td>ida</td>
<td></td>
</tr>
<tr>
<td>power</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Explicit Solvers</th>
<th>Implicit Solvers</th>
<th>ImEx Solvers</th>
</tr>
</thead>
<tbody>
<tr>
<td>euler</td>
<td>pvode</td>
<td>arkode</td>
</tr>
<tr>
<td>rk4</td>
<td>cvode</td>
<td>BDF multistep</td>
</tr>
<tr>
<td>rk3-ssp</td>
<td>petsc (various)</td>
<td></td>
</tr>
<tr>
<td>karniadakis</td>
<td>ida</td>
<td></td>
</tr>
<tr>
<td>power</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The default behaviour is to use either CVODE or IDA if present, otherwise use PVODE.
• 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:  On the command line
[solver]  ./executable
  type = ...  solver:type=...

• Usually best to try a few (ie RK4 and pvode) to find the optimal choice for your case
Some additional useful options are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>mxstep = 500</td>
<td>Number of timesteps to try before timestep is a failure</td>
</tr>
<tr>
<td>atol = 1e-10</td>
<td>Absolute tolerance used to determine error norm. Determines noise level of solution</td>
</tr>
<tr>
<td>rtol = 1e-5</td>
<td>Relative tolerances used to determine error norm. Indicates no of digits of relative accuracy for a single time step</td>
</tr>
<tr>
<td>adaptive = false</td>
<td>Use adaptive timestepping in rk4</td>
</tr>
<tr>
<td>use_precon = true</td>
<td>Use preconditioning in cvode and petsc</td>
</tr>
</tbody>
</table>
• To tell the time-solver to solve for a particular field we use the routine
  \[
  \text{bout\_solve}(n, \text{"density"});
  \]
  or the macro \[
  \text{SOLVE\_FOR}(n);
  \]
  or similarly for \(2 \leq n \leq 6\) \[
  \text{SOLVE\_FORn}(n,...);
  \]

• This calls a routine in the solver class
  \[
  \text{solver->add}(n,\text{"density"});
  \]
  which adds the variable to the state and residual vectors for input to the time-solver
• BOUT++ data is then be passed to the solver through a few protected functions:

\[
\begin{align*}
& \text{save-vars(BoutReal *udata);} \\
& \text{load-vars(BoutReal *udata);} \\
& \text{save-derivs(BoutReal *udata);} \\
& \text{load-derivs(BoutReal *udata);} \\
\end{align*}
\]
• 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.
• There are two mandatory functions that a time-solver implementation must contain

```cpp
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.
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…
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
PVODE: 3073 evals
PVODE: 75,585 evals

Central differencing with dissipation
Fromm (2\(^{nd}\) order)
XPPM (nonlinear limiter)
• For ImEx schemes the split operators must be defined

C style interface

```c
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

```cpp
class DriftWave : public PhysicsModel {
    int init(bool restart) {
        SetSplitOperator();
        ... 
    }

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

    int diffusive(BoutReal time) {
        // Implicit part
    }
};
```
"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
• Recall that the implicit time-solver(s) solves a newton iteration of the form

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

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

• It may be possible to define a new operator, $P$, such that

$$P(I - \gamma J)\delta f = -PG(f)$$

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

\[ P(I - \gamma J)\delta f = -PG(f) \]

\[ \tau \delta f = -(I - \gamma J)^{-1}G(f) \]
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)
1. Reduce equations

\[ \partial_t n = f(n, T) \]

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

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

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

\[ \partial_t n = 0 \]

\[ \partial_t T = \partial_{||} \chi_{||} \partial_{||} T \]
2. Calculate the Jacobian

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

For our reduced system

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

So

\[ I - \gamma J = \begin{pmatrix} 1 & 0 \\ 0 & 1 - \gamma \chi_0 T^{5/2} \partial_\|^2 \end{pmatrix} \]
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 $D = 1 - \gamma \chi_0 T_2^5 \partial_{||}^2$ so

$$
S^{-1} = \left(1 - \gamma \chi_0 T_2^5 \partial_{||}^2\right)^{-1}
$$
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 $S^{-1} = \left( 1 - \gamma \chi_0 T^2 \partial_{\parallel}^2 \right)^{-1}$
• 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
5. Write a precon function

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

```c
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)
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

```cpp
InvertPar *precon_inv;
int physics_init(bool restarting){
 ...
 precon_inv = InvertPar::create();
 precon_inv->setCoefA(1.0);
 ...
}
```
6. Apply preconditioner to time derivatives

Now in the precon function we apply the operator

```c
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");

}
```
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);
```
7. *Tell the solver to use the preconditioner*

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

<table>
<thead>
<tr>
<th>Solver</th>
<th>Setup</th>
<th>Wall time (s)</th>
<th>~Iteration count</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVODE</td>
<td>Isothermal</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>CVODE</td>
<td>Conduction removed</td>
<td>5</td>
<td>300</td>
</tr>
<tr>
<td>PVODE</td>
<td>None</td>
<td>221</td>
<td>37000</td>
</tr>
<tr>
<td>CVODE</td>
<td>None</td>
<td>276</td>
<td>45000</td>
</tr>
<tr>
<td>CVODE</td>
<td>BBD Preconditioner</td>
<td>58</td>
<td>17000</td>
</tr>
<tr>
<td>CVODE</td>
<td>Custom Preconditioner</td>
<td>9</td>
<td>800</td>
</tr>
</tbody>
</table>

Approximately 30x speedup with preconditioner
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
• 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