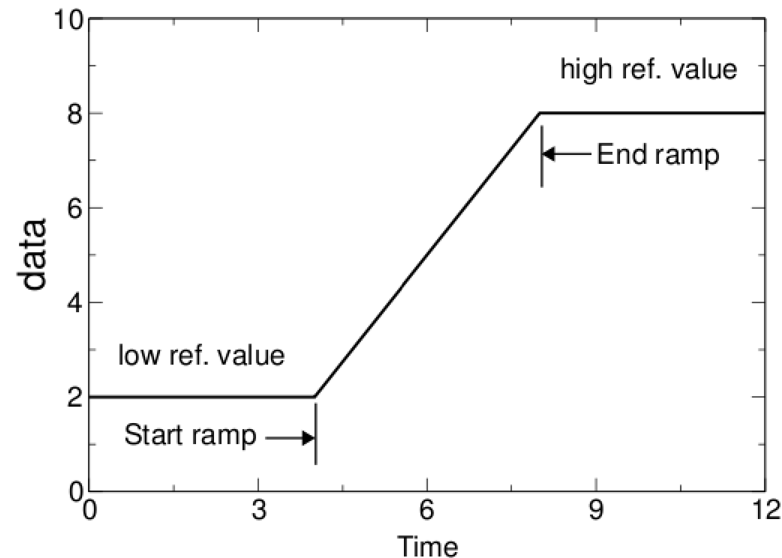
# Exercise: Implement a rampedFixedValue boundary condition (for OpenFOAM-2.1.x, and most likely 2.2.x)

# Sparse guidelines on how to implement a rampedFixedValue bc

- Pretend that you are in desperate need of a boundary condition that ramps the value of a variable from one value to another value within some specified time, i.e.:




- You have figured out that there is already a boundary condition that oscillates the value of a variable at a boundary, so you only need to change the time function of that boundary condition to get what you want:

  `$FOAM_SRC/finiteVolume/fields/fvPatchFields/derived/oscillatingFixedValue`

- You copy that to create a dynamic library according to what you have learnt in an excellent course...

# Sparse guidelines on how to implement a rampedFixedValue bc

You need to do the following:

- Copy the `oscillatingFixedValue` directory to an appropriate location outside the original installation, and rename `oscillating*` to `ramped*`
- Rename all the files from `oscillating*` to `ramped*`
- You use the `sed` command to efficiently substitute all `oscillating` to `ramped` in all the files (`*.H` and `*.C`). This changes the name of the class.
- Create a `Make` directory by looking at the original `Make` directory in `$FOAM_SRC/finiteVolume`. In `files`, you remove everything that has nothing to do with `oscillating*` and rename `oscillating` to `ramped`. (Make sure that you name the 'plural' file with 's'). You also remember that since you have moved the directory, you must add an include line to the `options` file, so that it can find the files in the `finiteVolume/lnInclude` directory. You add `my` at the start of the library name (after `lib`) to distinguish it from the original one, and make sure that it will be written to your user directory structure.
- At this point, you clean up the compilation (`wclean lib;rm -r Make/li*`) and compile the dynamic library (`wmake libso`) just to make sure that it compiles before doing any changes.

# Sparse guidelines on how to implement a rampedFixedValue bc

- You see that in `rampedFixedValueFvPatchField.H`, there are some private data defined

```
Field<Type> refValue_
autoPtr<DataEntry<scalar> > amplitude_
autoPtr<DataEntry<scalar> > frequency_
label curTimeIndex_
```

In your case, you instead need

```
Field<Type> refValueLow_
Field<Type> refValueHigh_
autoPtr<DataEntry<scalar> > startRamp_
autoPtr<DataEntry<scalar> > endRamp_
label curTimeIndex_
```

You make a note of which of your new private data are objects of the same class as the original private data. You realize that the original files have only one private data of type `Field<Type>`, while you need two.

# Sparse guidelines on how to implement a rampedFixedValue bc

- For each of the origial private data, you search the files and make sure that you have the same lines for your new private data that are of the same class as the original private data. In your case, you have to add some lines since you have two objects of `Field<Type>` instead of only one. You simply copy all lines that correspond to the original `refValue` and add the same lines just after the original ones, and then update to your new private data names. For the `operator==`, you for now just change to `refValueLow_`, and the same at `patchField = refValue_`... You change `amplitude` to `startRamp`, and `frequency` to `endRamp` everywhere.
- You clean and compile again, and realize that you don't get any error messages, which means that you didn't do any mistakes.
- Now you have all the structure ready, so you just need to implement the function.

# Sparse guidelines on how to implement a rampedFixedValue bc

- You change in rampedFixedValueFvPatchField.C the operator== where you temporarily just changed refValue to refValuelow, to:

```
fixedValueFvPatchField<Type>::operator==
(
    refValueLow_ + (refValueHigh_ - refValueLow_)*currentScale()
);
```

  The function currentScale() is the ramp fraction at time t, which also needs to be modified...

- You change the currentScale() function to:

```
return
    min
    ( 1.0, max (
               (this->db().time().value() - a)/
               (f - a), 0.0));
```

- In function updateCoeffs(), you change the evaluation of the patchField to:

```
patchField = refValueLow_
             + (refValueHigh_ - refValueLow_)*currentScale();
```

- You clean an compile, and you are done!

# A few comments

- The `write` function should now look like this:

```
template<class Type>
void rampedFixedValueFvPatchField<Type>::write(Ostream& os) const
{
    fixedValueFvPatchField<Type>::write(os);
    refValueLow_.writeEntry("refValueLow", os);
    refValueHigh_.writeEntry("refValueHigh", os);
    os.writeKeyword("offset") << offset_ << token::END_STATEMENT << nl;
    startRamp_->writeData(os);
    endRamp_->writeData(os);
}
```

This function makes sure that all the information of the boundary condition is written in the output time directories. This useful when the simulation is restarted from the latest time.

# A few comments

- The generic `rampedFixedValueFvPatchField<Type>` class becomes specific for scalar, vector, tensor, ... by using the command (see the `*Fwd*` file):
  `makePatchTypeFieldTypedefs(rampedFixedValue)`
- This function is defined in `$FOAM_SRC/finiteVolume/fvPatchField.H` and it uses typedef for this purpose:

```
typedef rampedFixedValueFvPatchField<scalar> rampedFixedValueFvPatchScalarField;
typedef rampedFixedValueFvPatchField<vector> rampedFixedValueFvPatchVectorField;
typedef rampedFixedValueFvPatchField<tensor> rampedFixedValueFvPatchTensorField;
```

# A few comments

- It adds to the `runTimeSelectionTable` the new boundary conditions created in `rampedFixedValueFvPatchFields.H`, by calling the function: `makePatchTypeFieldTypedefs(rampedFixedValue);`
- In this way, the new boundary condition can be used for `volScalarField`, `volVectorField`, `volTensorField`, ... just typing in the field file:

```
boundaryField
{
    inlet
    {
        type rampedFixedValue;
        refValueLow uniform 10;  // example for a volScalarField
        refValueHigh uniform 20; // example for a volScalarField
        startRamp 20;
        endRamp 50;
        value uniform (0 0 0); //Dummy for paraFoam
    }
}
```

- So, you can now easily use the `rampedFixedvalue` boundary condition for the cavity case, with the original icoFoam solver.