Turbojet driver

This note presents a brief documentation of the external class Thrust which is used as a driver model of single-flow turbojet.

A driver can coordinate recalculation of a project Thermoptim under special rules. For our model, which is presented in the form of guidance pages for practical work on the turbojet, the driver can coordinate updates to the diffuser, the nozzle and the compression ratio in the whole project, to calculate the values of specific thrust and consumption per unit of thrust, which are not provided directly by Thermoptim, and perform sensitivity studies by saving the results to a file. Operating in this way, it greatly facilitates the use of the model.

As explained in details in Volume 3 of the reference manual, there are two ways to control Thermoptim: either totally from an external application that instantiates the software and sets the parameters, or partially, for a given project. It is in this latter context that we are.

In this case, we associate a specific management class to the model, and this class is instantiated when loading the project. We must first make an external class for it to be loaded into Thermoptim at launch. Once the project is open, the selection of the driver class associated with it is done through item "Driver screen" of menu Special of the simulator, as described at the end of this note. When writing the project file, the name of the driver class is saved so it can be instantiated during a subsequent loading of the project.

Brief recall of the features of the model to drive

Figure 1 shows a synoptic view of the turbojet model that is to be controlled. It involves an inlet diffuser for creating a dynamic pressure at the compressor inlet when the aircraft is in flight, a gas generator comprising a compressor, a combustion chamber and turbine, and finally a nozzle which propels the aircraft.

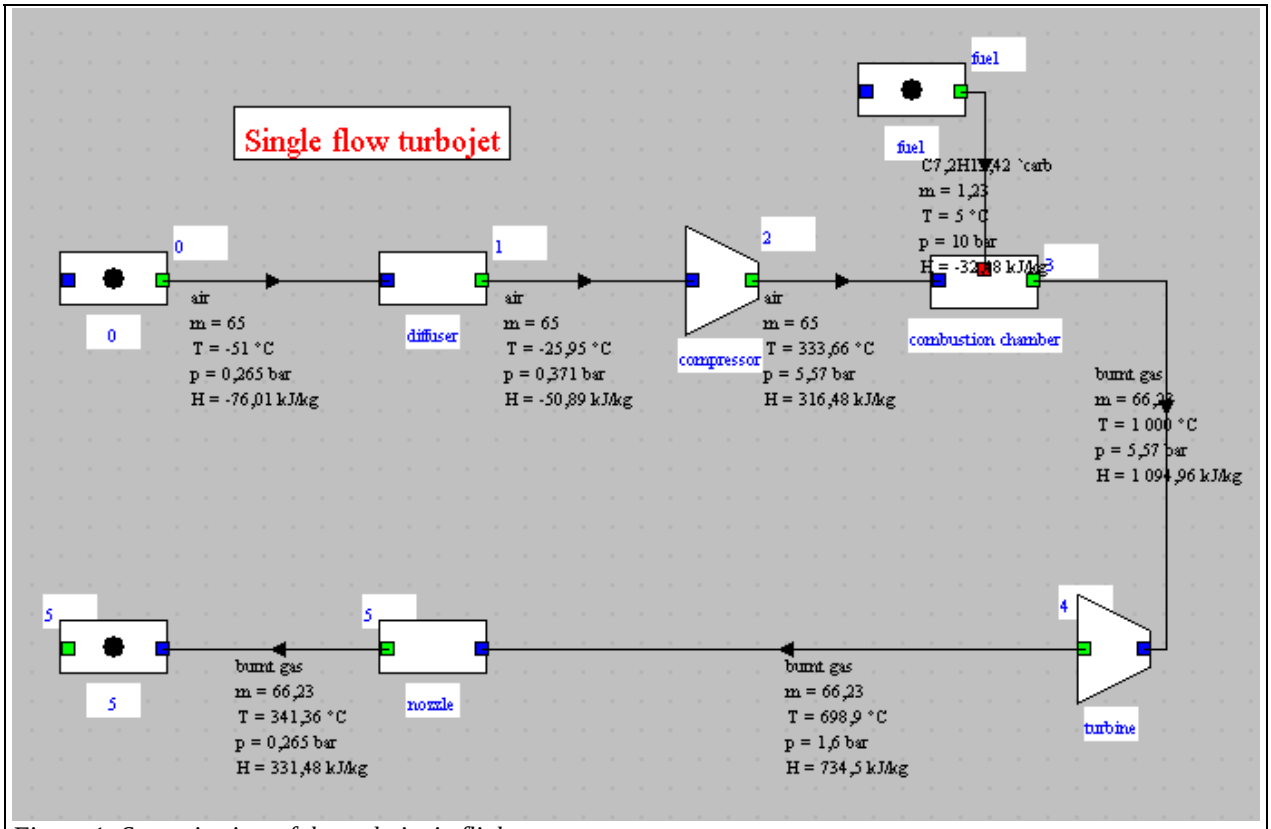

Figure 1: Synoptic view of the turbojet in flight

To calculate an operating point, we must operate as follows:

- Enter in the diffuser screen values of external conditions, that is to say, the aircraft speed (m/s), pressure and ambient temperature
- Enter in the nozzle screen the value of ambient pressure
- Set the compression ratio
- Enter in the combustion screen the turbine inlet temperature
- Recalculate the entire project with this setup
- Get in the screens of the diffuser and the nozzle the values of aircraft C_0 and exhaust gas C_5 speeds and flows sucked $\dot{m}_0$ and rejected $\dot{m}_5$ to calculate the thrust and specific consumption per unit of thrust as shown below

The expression of the thrust is $F = \dot{m}_0 C_0 - \dot{m}_5 C_5$, and the specific thrust is $F/\dot{m}_0$.

The mass flow of fuel is $\dot{m}_c = \dot{m}_5 - \dot{m}_0$, and consumption per unit thrust is $\dot{m}_c /F$.

Such a sequence of operations is tedious to repeat when you want to do sensitivity studies, and so contains errors. The completion of a driver is perfectly justified in this case.

Presentation of the external class

The driver class is an extension of extThopt.ExtPilot, which derives from rg.thopt.PilotFrame. Figure 2 shows a screen for the driver, with above the definition of the conditions outside the aircraft, in the middle a field to enter a value of the compression ratio to perform a single calculation, and in the bottom entry of compression ratio bounds to perform several calculations (10 in this case) and save the results.

We will not present here the construction of the GUI, which poses no particular problem and whose code is entirely conventional.
The sequence of initialization and calculation is as follows:

1) we look for instances of external classes brought into play. Classes Diffuser and Nozzle were declared globally, Nozzle as an array of dimension 2 so that the code can be easily modified to serve as a driver in a turbofan. A consistency check is done on the number of instances of classes. It could be improved.

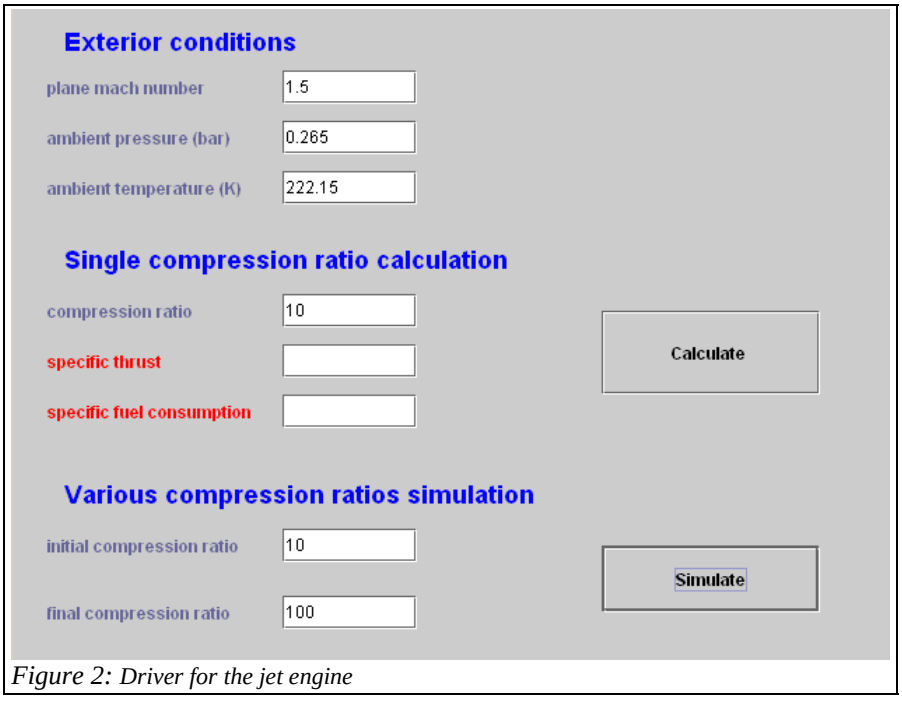

Figure 2: Driver for the jet engine

```
  void setupProject(){
      //on récupère ici les instances des transfos externes dont on a besoin
      //afin d'accéder à leurs différents paramètres pour les calculs ultérieurs
      //we get the instances of external processes needed in order to
      //access their settings for subsequent calculations
      nozzle=new Nozzle[2];
      nomNozz=new String[2];
      nomNozz[0]="";
      nomNozz[1]="";
      Vector vExt=proj.getExternalClassInstances();//Vector contenant les classes externes
                                                   //Vector containing external classes
      int k=0,j=0;
      for(int i=0;i<vExt.size();i++){
          Object[] obj=new Object[6];
          obj=(Object[])vExt.elementAt(i);
          ExtProcess ep=(ExtProcess)obj[1];
          if(ep instanceof Diffuser){
              diff=(Diffuser)ep;
              nomDiff=diff.getName();
              j++;
          }
          if(ep instanceof Nozzle){
              nozzle[k]=(Nozzle)ep;
              nomNozz[k]=nozzle[k].getName();
              k++;
          }
      }
      if(j*k==1)isBuilt=true;//test de cohérence du pilote par rapport au modèle
                             //driver consistency test
}
```

2) other initializations call methods getProject() and proj.getEditorComponentList() presented in Volume 3 of the reference manual, which provide the reference of the project (proj) and the list of components in the diagram editor (method setupProject() is presented above).

```
public void init(){
    isInitialized=true;
    proj=getProjet();
    setupProject();
    //On récupère la liste et le type des composants présents dans l'éditeur de schémas
    //Gets the list and type of components in the diagram editor
    String[]listComp=proj.getEditorComponentList();
    composant=new String[listComp.length];
    nomComposant=new String[listComp.length];

    //on en extrait les noms des transfos du noyau dont on a besoin
    //à savoir le compresseur et la chambre de combustion
    //we get the names of processes needed (i.e. the compressor and the combustion chamber
    for(int i=0;i<listComp.length;i++){
        composant[i]=Util.extr_value(listComp[i], "type");
        nomComposant[i]=Util.extr_value(listComp[i], "name");
        if(composant[i].equals("Combustion"))combustionChamberName=nomComposant[i];
        if(composant[i].equals("Compression"))compressorName=nomComposant[i];
    }

    //test de cohérence (des messages d'erreur plus précis seraient souhaitables)
    //consistency tests (more specific error messages would be desirable)
    if((!combustionChamberName.equals(""))&&(!compressorName.equals(""))&& isBuilt)isBuilt=true;
    if(isBuilt)setVisible(true);//on n'ouvre le pilote que si sa structure est correcte
    //the driver is opened only if the structure is appropriate
    //initialisations pour la simulation (calculs pour 10 rapports de compression)
    //initializations for simulation (calculation forr 10 compression ratios)
    thrust=new double[10];
    conso=new double[10];
    taux=new double[10];
}
```

3) initialization and calculation for a single compression ratio (the code to perform the simulation for 10 compression ratios is very similar, with the added backup of the results in a file). Warning: the new setting of the compression is done by modifying the pressure downstream of the point, which requires that the compression is carried out with a ratio "calculated" and not "set" as in the project without a driver.

```
  void bCalculate_actionPerformed(java.awt.event.ActionEvent event){
      if(!isInitialized)init();//First initialization
      Pext=Util.lit_d(ambiantP_value.getText());//pression extérieure / ambient pressure (bar)
      Text=Util.lit_d(ambientT_value.getText());//température extérieure / ambient temperature (K)
      diff.setAmbientConditions(Text, Pext);//mise à jour du point amont du diffuseur / diffuser update
      nozzle[0].setExternalPressure(Pext);//mise à jour de la pression en aval de la tuyère / nozzle update
      Ma=Util.lit_d(planeMach_value.getText());//Mach de l'avion / plane Mach number
      double C=diff.getSpeed(Ma, Text, Pext);//vitesse en amont du diffuseur / velocity at the diffuser inlet
      diff.setInletVelocity(C);
      double comprRatio=Util.lit_d(comprRatio_value.getText());//rapport de compression / compression ratio
      for(int i=0;i<3;i++){//on effectue 3 recalculs pour garantir la convergence
                           //three calculations are performed to ensure convergence
          if(!combustionChamberName.equals("")){//mise à jour du compresseur
              String[] args=new String[2];
              args[0]="process";
              args[1]=compressorName;
              Vector vProp=proj.getProperties(args);
              String amont=(String)vProp.elementAt(1);
              String aval=(String)vProp.elementAt(2);
              args[0]="point";
              args[1]=amont;
              vProp=proj.getProperties(args);
              Double f=(Double)vProp.elementAt(3);
              double Pamont=f.doubleValue();

              double Paval=Pamont*comprRatio;//mise à jour du point aval du compresseur
                                             //update of the compressor outlet point
              Vector vPoint=new Vector();
              vPoint.addElement(aval);
              vPoint.addElement("false");//update temperature
              vPoint.addElement("0");
              vPoint.addElement("true");//update pressure
              vPoint.addElement(Util.aff_d(Paval));
              vPoint.addElement("false");
              vPoint.addElement("1");
              proj.updatePoint(vPoint);
          }
             proj.calcThopt();
        }
        //calcul des performances globales du moteur
        //overall turbojet performance
        double Cin=diff.getInletVelocity();
        double Cout0=nozzle[0].getOutletVelocity();
        double airFlow=diff.getFlow();
        double combustionFlow=airFlow;
        if(!combustionChamberName.equals("")){
            String[] args=new String[2];
            args[0]="process";
            args[1]=combustionChamberName;
            Vector vProp=proj.getProperties(args);
            Double f=(Double)vProp.elementAt(3);
            combustionFlow=f.doubleValue();
        }
        double fAR=combustionFlow/airFlow-1;//fuel air ratio
        double specThrust=((1+fAR)*Cout0-Cin)/1000;
        thrust_value.setText(Util.aff_d(specThrust, 3));
        double specFuelCons=fAR/specThrust;
        fuel_value.setText(Util.aff_d(specFuelCons, 5));
    }
}
```

Loading the Driver

To load the driver, select item "Driver frame" of menu "Special" of the simulator. A combo then offers a list of available drivers (Figure 3). Select the one you want to load (here "thrust") and validate.

When a driver is already loaded, item "Driver frame" of menu "Special" of the simulator opens the screen in Figure 4, which allows you to return to the driver loaded (line with its name), to select another, or delete the existing one without replacing it. Only one driver may be associated with a project.

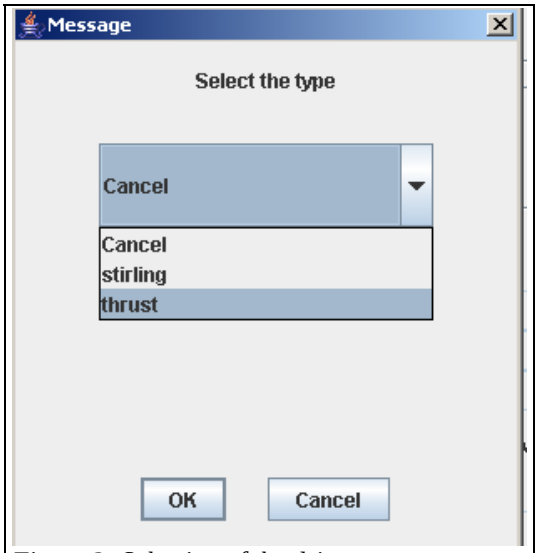

Figure 3: Selection of the driver

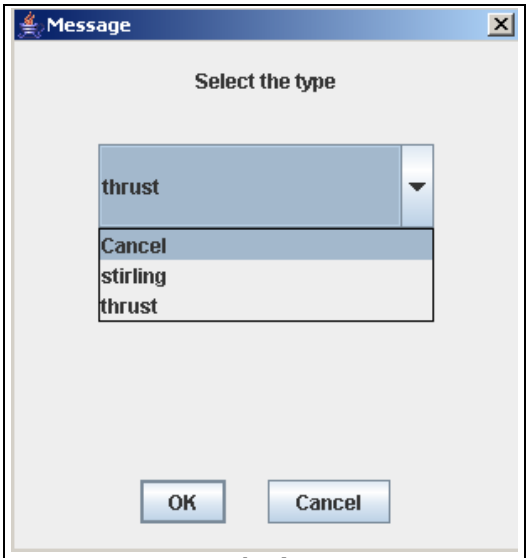

Figure 4: Removing the driver