STIRLING ENGINE DRIVER

In this note, we present a pilot that calculates the energy balance of the solar Stirling engine (Figure 1).

It is indeed a very simple example that allows one to introduce this type of external class without having to dwelve into the details of a complex thermodynamic model.

A Stirling engine is a special type of engine, which works in a closed system, and implements cooled compression and heated expansion, so that Thermoptim's usual performance indicators cannot be used directly: purchased energy is the sum of the heat supplied to the hot source (in this example a solar concentrator) and that supplied during the expansion.

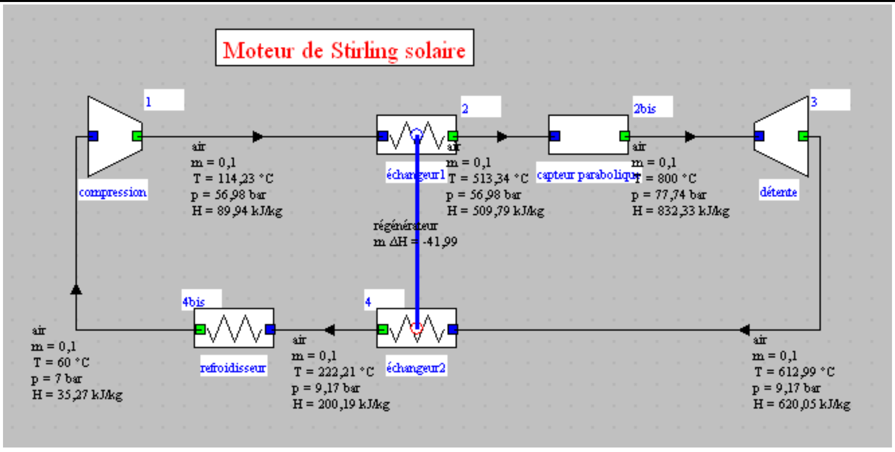

Figure 1 : Solar Stirling Engine

The objective we assign to the pilot is to draw up a summary table of the energies involved in the engine, in the form of heat or mechanical power, and to calculate the cycle efficiency. Figure 2 shows the type of screen that can be imagined.

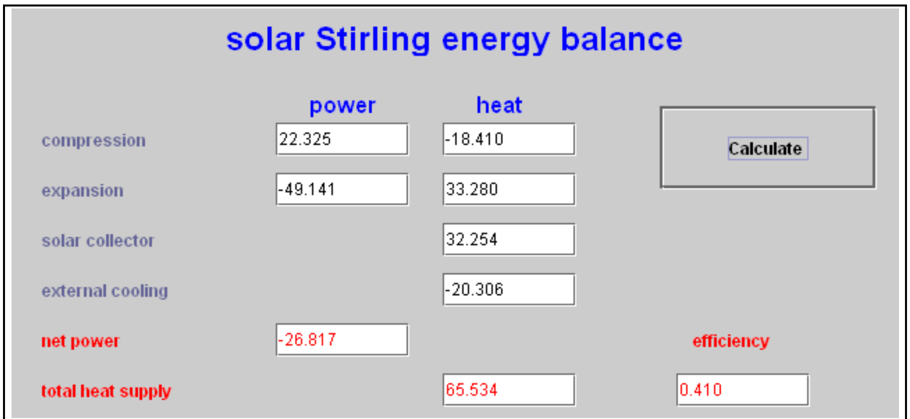

Figure 2 : Pilot Screen

CREATION OF THE CLASS, VISUAL INTERFACE

To create an external driver, simply subclass extThopt. ExtPilot.

The realization of the visual interface does not pose any particular problem and we will not comment on it here.

The constructor must end with a String specifying the code that will designate the driver from the list of available ones: type = "stirling";

It is also recommended to document the class:
public String getClassDescription () {
return "pilot for a simple Stirling motor \ n \ nauthor: R. Gicquel february 2008";}

RECOGNITION OF COMPONENT NAMES

It is possible to automatically recognize the names of the various components constituting the model, sorting them by type, which gives the pilot a greater genericity than if these names are entered as a String in the code. This is the purpose of the init () and setupProject () methods, which use methods allowing Thermoptim to access the component names of the diagram editor and the list of available external classes (for the external process representing the solar concentration collector).

```
public void init(){
isInitialized=true;
proj=getProjet();
setupProject();
//On récupère la liste et le type des composants présents dans l'éditeur de schémas
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
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("Compression"))compressorName=nomComposant[i];
    if(composant[i].equals("Expansion"))expansionName=nomComposant[i];
}
//test de cohérence (des messages d'erreur plus précis seraient souhaitables)
if((!expansionName.equals(""))&&(!compressorName.equals(""))&& isBuilt)isBuilt=true;
if(isBuilt)show();//on n'ouvre le pilote que si sa structure est correcte
//initialisations pour la simulation (calculs pour 10 rapports de compression)
}

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
    Vector vExt=proj.getExternalClassInstances();//Vector contenant les classes externes
    int 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 SolarConcentrator){
            collector=(SolarConcentrator)ep;
            collectorName=collector.getName();
            j++;
        }
    }
    if(j==1)isBuilt=true;//test de cohérence du pilote par rapport au modèle
}
```

CALCULATIONS CARRIED OUT AND DISPLAY

Once the names of the various components have been identified, their properties are accessed through the project's getProperties () method, which allows you to get all the values you need.

```
void bCalculate_actionPerformed(java.awt.event.ActionEvent event){
    if(!isInitialized)init();//la première fois, on initialise, car il faut un constructeur sans argument
                            //pour instancier la classe par le RMI

    String[] args=new String[2];
    args[0]="process";
    args[1]=compressorName;
    Vector vProp=proj.getProperties(args);
    String amont=(String)vProp.elementAt(1);//point amont (non utilisé ici)
    String aval=(String)vProp.elementAt(2);//point aval (non utilisé ici)
    Double f=(Double)vProp.elementAt(4);
    double deltaUcompr=f.doubleValue();//puissance compresseur
    f=(Double)vProp.elementAt(12);
    double Qcompr=f.doubleValue();//chaleur compresseur
    args[1]=expansionName;
    vProp=proj.getProperties(args);
    amont=(String)vProp.elementAt(1);
    aval=(String)vProp.elementAt(2);
    f=(Double)vProp.elementAt(4);
    double deltaUexpan=f.doubleValue();//puissance détente
    f=(Double)vProp.elementAt(12);
    double Qexpan=f.doubleValue();//chaleur détente
    args[0]="process";
    args[1]=collectorName;
    vProp=proj.getProperties(args);
    f=(Double)vProp.elementAt(4);
    double solarHeat=f.doubleValue();//chaleur solaire

    //calcul des performances globales du moteur et affichages
    tauExpan_value.setText(Util.aff_d(deltaUexpan, 3));
    netPower_value.setText(Util.aff_d(deltaUexpan+deltaUcompr, 3));
    tauCompr_value.setText(Util.aff_d(deltaUcompr, 3));
    Q_value.setText(Util.aff_d(Qexpan+solarHeat, 3));
    eta_value.setText(Util.aff_d((-deltaUexpan-deltaUcompr)/(Qexpan+solarHeat), 3));
    expanHeat_value.setText(Util.aff_d(Qexpan, 3));
    comprHeat_value.setText(Util.aff_d(Qcompr, 3));
    solarHeat_value.setText(Util.aff_d(solarHeat, 3));
    extCooling_value.setText(Util.aff_d(-(Qexpan+solarHeat+deltaUexpan+deltaUcompr+Qcompr), 3));
}
```