Etude et optimisation des trajectoires d'une fusée hydropneumatique

TIPE

2023 - 2024

- Introduction
- Etude Théorique
- Approche Informatique
- Expérimentations et simulation
- Conclusion

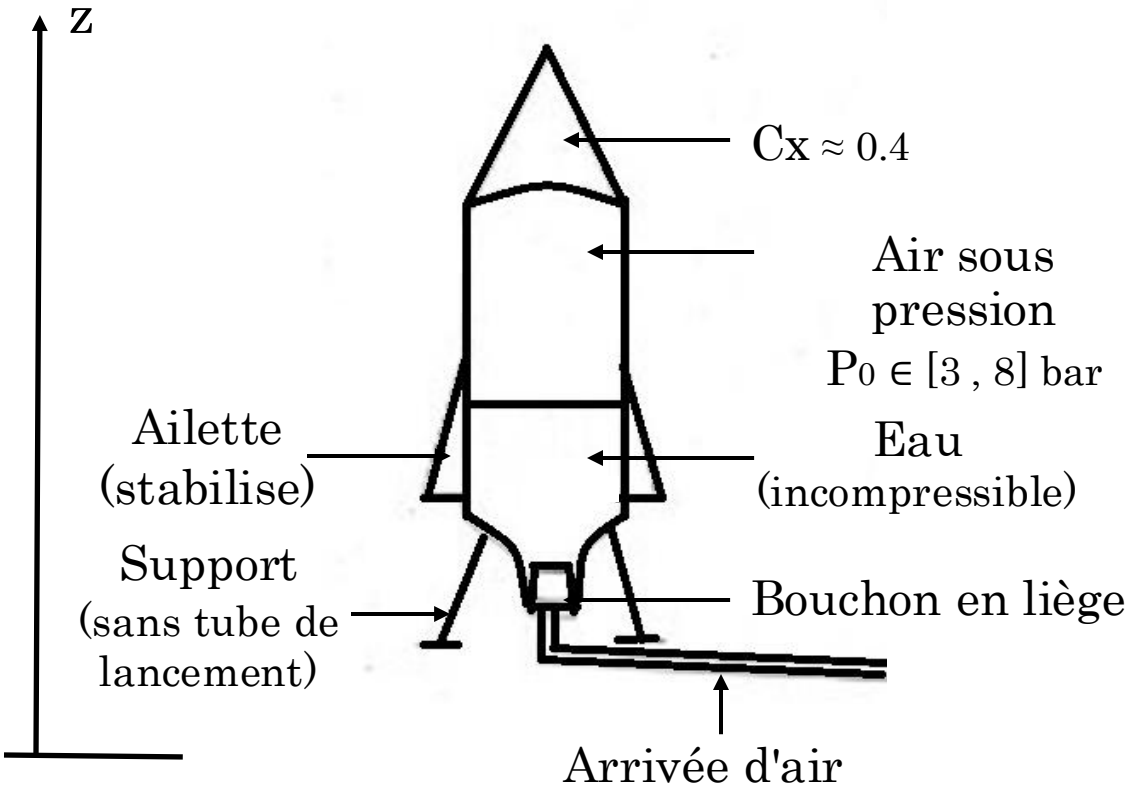
Figure 1 - Fusée utilisée durant l'étude

Problématique :

Le mouvement des fusées à eau étant soumis à de nombreux paramètres, comment les prendre en compte et les optimiser afin d'obtenir une trajectoire proche de celle voulu ?

Modèle Théorique

Figure 2 - Modèle considéré



Goulot droit

Figure 3 - Type de goulot utilisé

Phase d'expulsion de l'eau

Trainée de la fusée : $-\frac{1}{2} \cdot \rho_{air} \cdot C_x \cdot S \cdot \vec{v} \cdot v$
Poids : $- \ m(t) \cdot g$
Frottements dans le goulot

Poussée : **?**

La conservation de la masse conduit à : $\frac{dm}{dt} = \rho_{eau} \cdot S_{out} \cdot u$

Système Ouvert

Equation de Navier - Stokes :

$$\frac{d}{dt}\left(m_b \cdot \vec{v} + \int \rho_{eau} \cdot \left(\vec{u} + \vec{v}\right) . dV\right) = (P_{out} - P_a) \cdot A_{out} + \frac{dm}{dt} \cdot \vec{u} + \overrightarrow{F_{frot}} + m(t) \cdot \vec{g} + \frac{dm}{dt} \cdot \left(\overrightarrow{u_{out}} + \vec{v}\right)$$

On obtient en développant à gauche :

$$m(t) \cdot \vec{a} = (P_{out} - P_a) \cdot A_{out} + \frac{dm}{dt}\vec{u} + \overrightarrow{F_{frot}} + m(t) \cdot \vec{g} - \frac{d}{dt}\left(\int \rho_{eau} \cdot \vec{u} \cdot dV\right)$$

Simplification et dentification :

$$\overrightarrow{F_{poussee}} = \frac{dm}{dt}\vec{u} = \rho_{eau} \cdot S_{out} \cdot u \cdot \vec{u}$$

Détermination de u avec l'équation de Bernoulli (1) :

$$\frac{du}{dt} = -\frac{\gamma \cdot P_0 \cdot {V_0}^{\gamma} \cdot S_{out}}{\rho_{eau}} \cdot \left(\frac{\frac{\rho_{eau} \cdot u^2}{2} + P_a}{P_0 \cdot {V_0}^{\gamma}}\right)^{\frac{\gamma+1}{\gamma}}$$

(1) Admis : Cf annexe pour énoncé et conditions d'application

Densité dans la bouteille : $\frac{d\rho_t}{dt} = \frac{dm}{dt} \cdot \frac{1}{V_b}$

Nombre de Mach : $M = \frac{|u|}{c}$

Pression de l'air dans la bouteille : $P_t(t) = P_1 \cdot \left(1 + \frac{t}{\tau}\right)^{\frac{2\cdot\gamma}{1-\gamma}}$

Constante : $\tau = \frac{2 \cdot V_b}{c_2 \cdot S_{out} \cdot (\gamma - 1)} \cdot \left(\frac{\gamma + 1}{2}\right)^{\frac{\gamma+1}{2\cdot(\gamma-1)}}$

On considère que la propulsion est sonique :

L'impulsion s'arrête lorsque $P = \beta \cdot P_a$ où β est un coefficient dépendant du gaz ambiant

Phase d'expulsion de l'air ou "gas-impulse"

Force de poussée en découlant :

$$F_{pousse} = 2 \cdot S_{out} \cdot P_t \cdot \left(\frac{2}{\gamma + 1} \right)^{\frac{1}{\gamma - 1}} - P_a \cdot S_{out}$$

Obtention de l'impulsion en intégrant :

$$I = \frac{P_1 \cdot V_b}{c_2} \cdot \sqrt{\frac{8}{\gamma + 1}} \left[1 - \left(\frac{\beta \cdot P_a}{P_1} \right)^{\frac{\gamma+1}{2 \cdot \gamma}} + \frac{P_a \cdot \left(\frac{\gamma+1}{2} \right)^{\frac{\gamma}{\gamma-1}}}{P_1 \cdot (\gamma - 1)} \cdot \left(1 - \left[\frac{P_2}{\beta \cdot P_a} \right]^{\frac{\gamma-1}{2 \cdot \gamma}} \right) \right]$$

Principe fondamental de la dynamique :

$$m_b \cdot a = F_{poussee} + F_{drag} - m_b \cdot g$$

La vitesse augmente d'environ $\frac{I}{m_b}$

PFD : $$\frac{dv}{dt} = -g \mp \frac{1}{2 \cdot m_b} \cdot C_x \cdot \rho_{air} \cdot S \cdot v^2$$

Soit : $$\frac{1}{g} \cdot \frac{dv}{dt} = -1 \mp \left(\frac{v}{v_t}\right)^2$$

On se ramène au système : $$v(t) = \begin{cases} v_t \cdot \tan\left(\frac{g\,(t_{top} - t)}{v_t}\right) & \text{monté} \\ v_t \cdot \tanh\left(\frac{g\,(t_{top} - t)}{v_t}\right) & \text{descente} \end{cases}$$

Connaître les paramètres initiaux de cette phase permet de déterminer toutes les autres données (2)

Phase de vol

(2) Cf annexe pour toutes ces expressions

Modèles d'approximation de l'expulsion de l'eau

1er modèle :

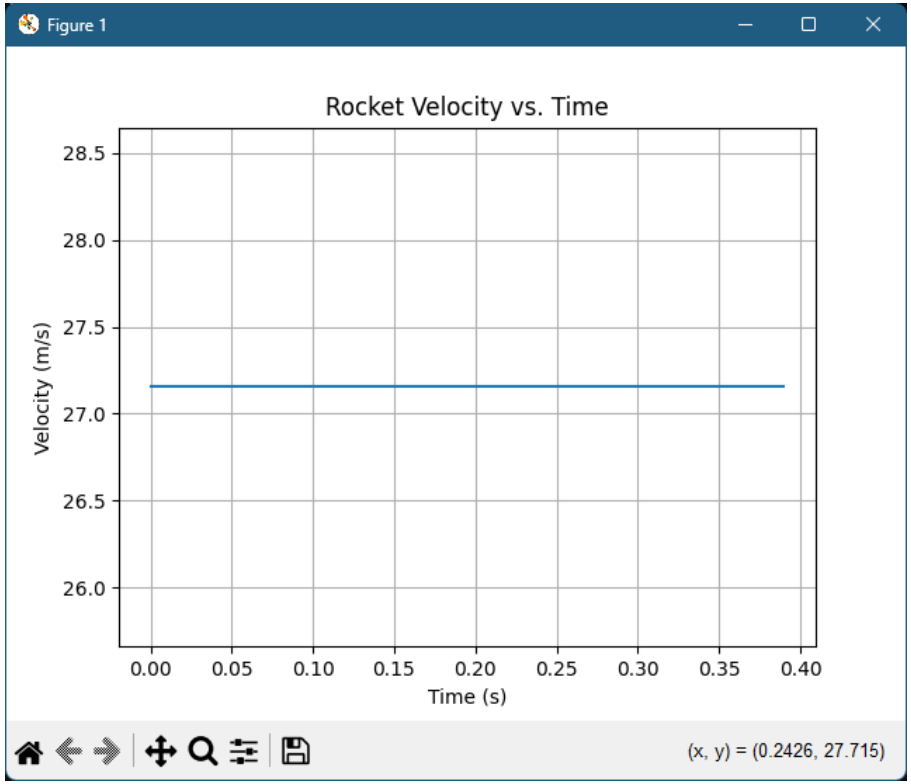


Figure 4 - Résultat avec u considéré constant

On considère :

$$u = \sqrt{\frac{2 \cdot (P_1 - P_a)}{\rho_{eau}}}$$

2ème modèle :

```
Entrée : Une équation différentielle y
         Un intervalle de temps dt
         Un temps final tf
         Un temps initial t0
         Une condition initial y0

Sortie : Un tableau t contenant tous les temps
         Un tableau y_tab contenant les solutions de y en fonction du temps

Effet de bord : Aucun

Runge_Kutta_4(y, y0, t0, tf, dt) :
    t[] ← [t0, t0 + dt, ..., tf]
    y_tab[] ← [0, ..., 0] de même longueur que t

    Pour i allant de 0 à longueur(t) Faire :
        k1 ← dt * y(t[i-1], y_tab[i-1])
        k2 ← dt * y(t[i-1] + 0.5*dt, y_tab[i-1] + 0.5*k1)
        k3 ← dt * y(t[i-1] + 0.5*dt, y_tab[i-1] + 0.5*k2)
        k4 ← dt * y(t[i-1] + dt, y_tab[i-1] + k3)
        y_tab[i] ← y_tab[i-1] + (k1 + 2*k2 + 2*k3 + k4) / 6

    Renvoyer t et y_tab
```

Algorithme de Runge Kutta d'ordre 4

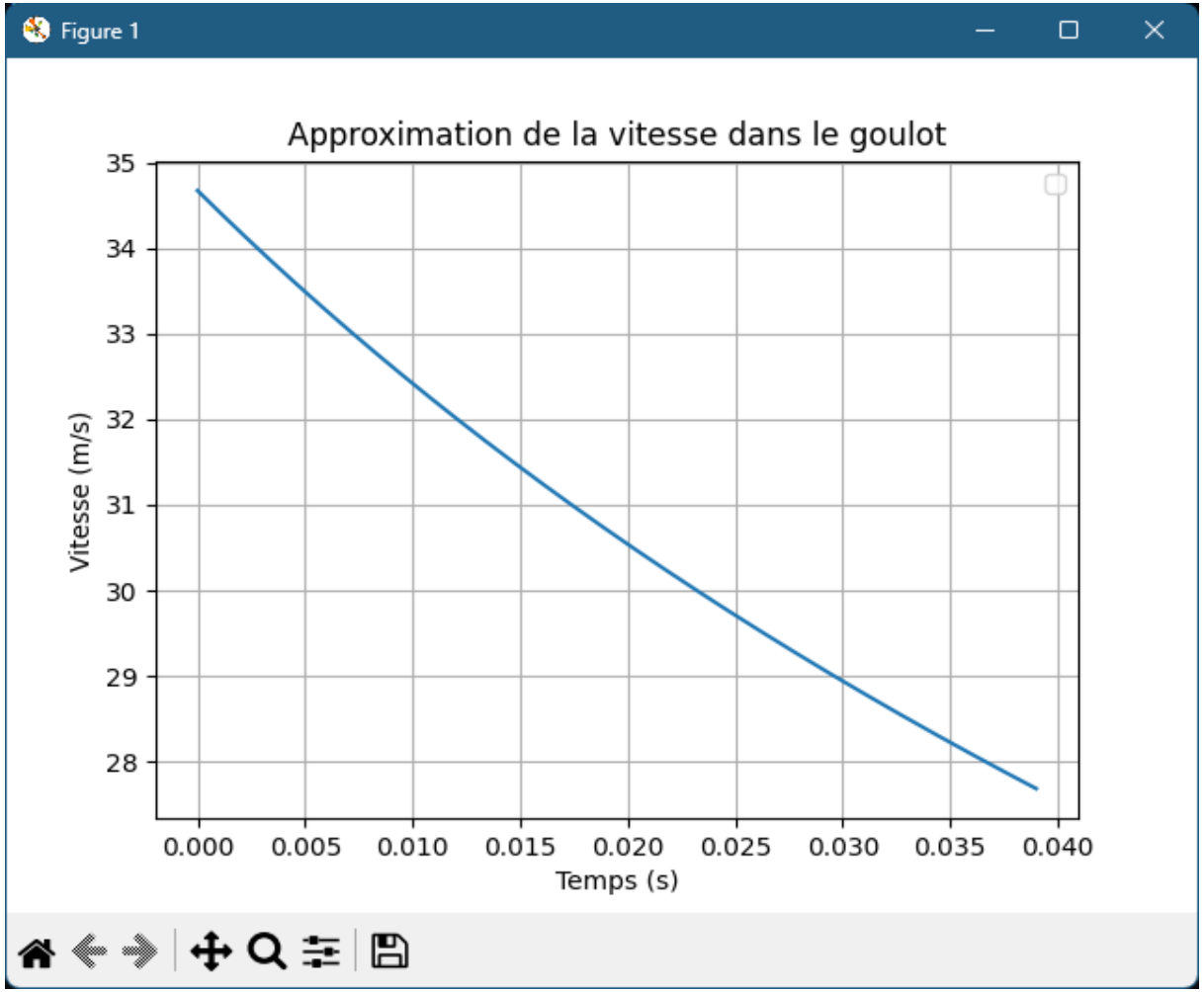


Figure 5 - Résultat avec méthode de Runge Kutta

Phase de propulsion :

Burnout	
Time	**0.08 s**
Altitude	**4.11 m**
Velocity	**76.79 m/s**
Acceleration	**-14.97 G**

Mise au point du simulateur

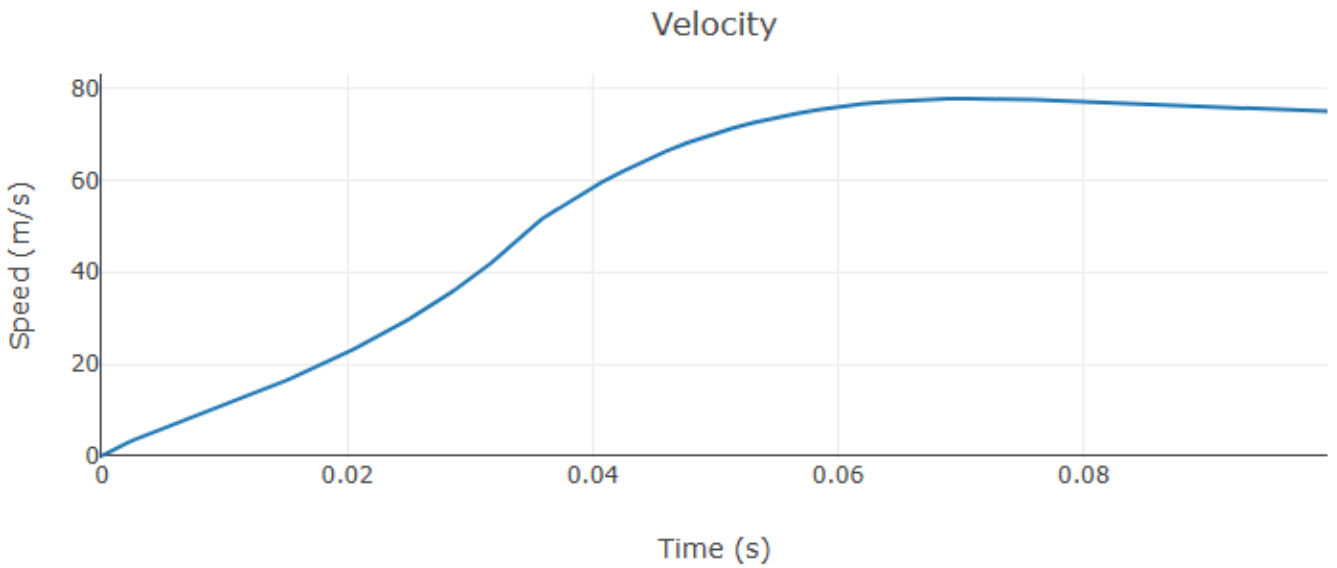


Figure 6 - Résultats de post poussée renvoyés par un simulateur tiers (aircommandrockets.com)

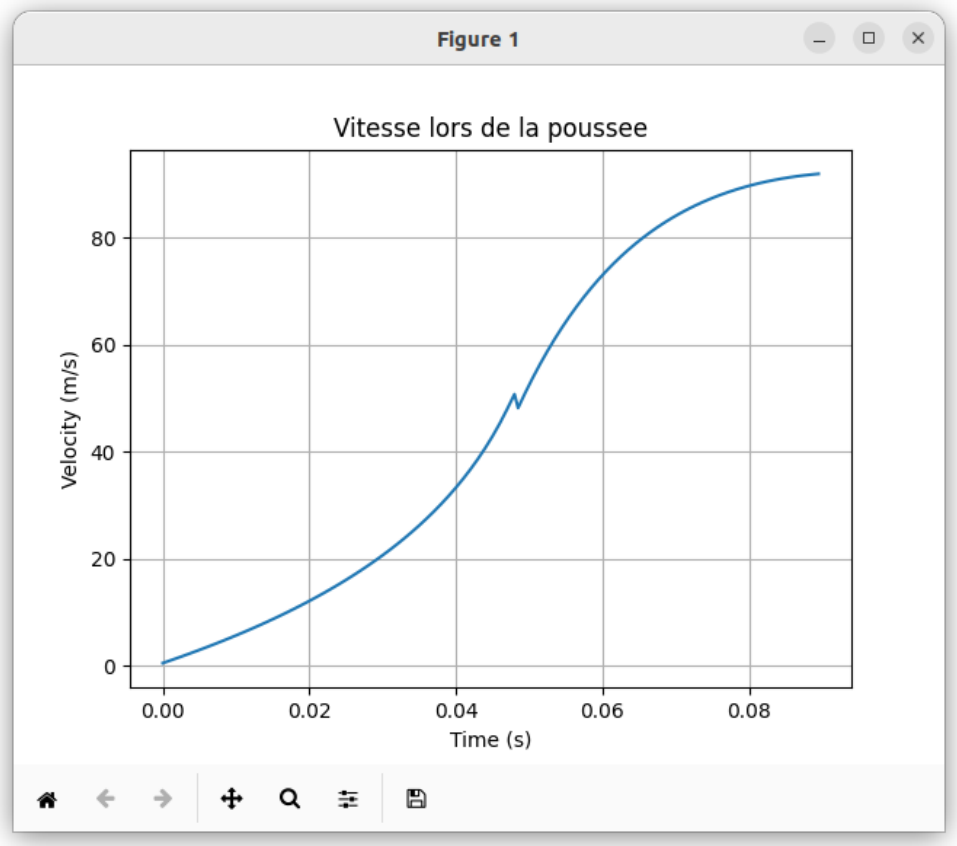


Figure 7 - Tracé des résultats de post poussée par un programme Python

```
justin@justin-linux:~/Bureau/TIPE$ ./exec

Vitesse à la fin de la poussée = 73.296395

Temps à la fin de la poussée  = 0.081343

Altitude à la fin de la poussée  = 5.519297
justin@justin-linux:~/Bureau/TIPE$
```

Figure 8 - Résultats de post poussée renvoyés par un programme C

Données de fin de vol :

Apogee	
Time	**2.82 s**
Altitude	**62.33 m (204 feet)**

Crashdown	
Time	**7.28 s**

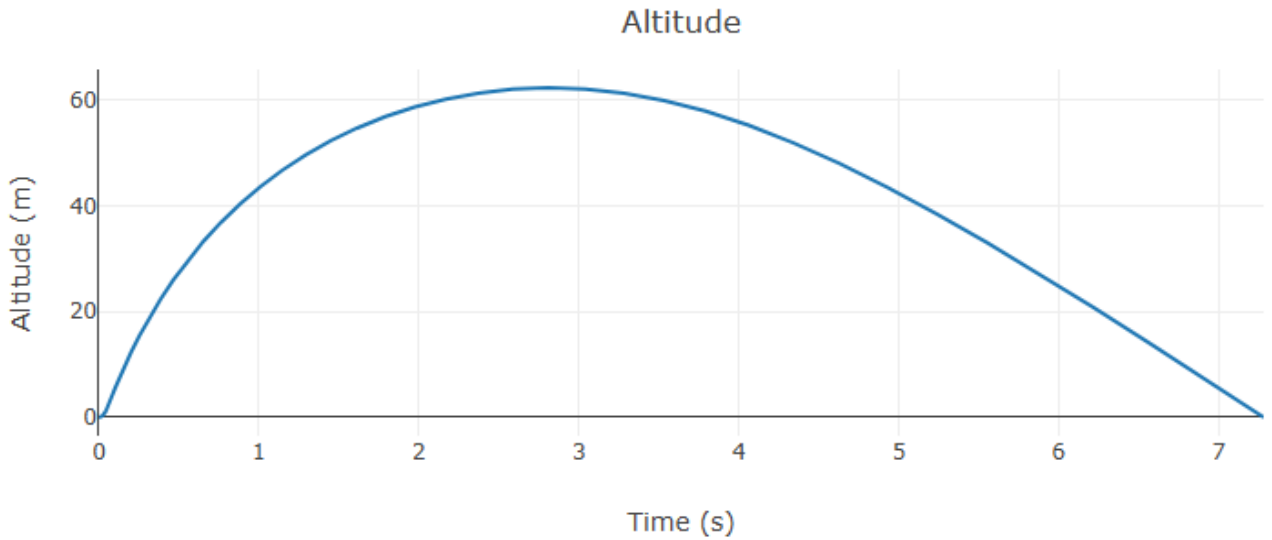


Figure 9 - Résultats pour l'altitude renvoyés par un simulateur tiers (aircommandrockets.com)

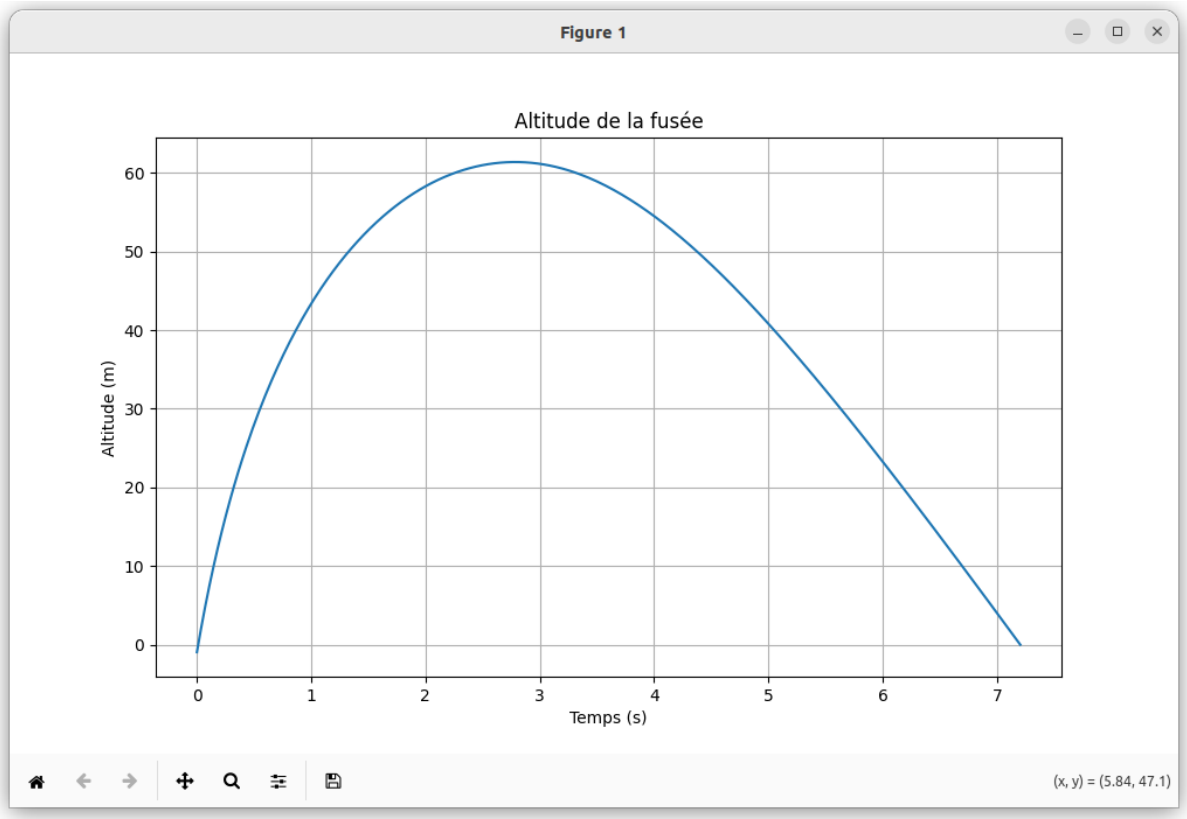


Figure 10 - Tracé de l'altitude fait par un programme python (identique au programme en C)

Atteindre l'altitude voulu :

```
RETROUVE_ALTITUDE (altitude, bouteille, environnement):
    res ← fusee ayant les parametres de bouteille
    res.water_volume ← 0
    res.pressure ← 0
    Tant que (calcul_z_top(res, environnement) ≠ altitude) Faire :
        res.pressure ← pression aleatoire entre 3 et 8 bars
        res.water_volume ← volume aleatoire entre 0.0002 et 0.0015 m^3
    Renvoyer res
```

Récupération des paramètres initiaux

```
Volume d'eau à mettre (en m³) : 0.000800

Pression dans la bouteille (en Pa) : 759000

Altitude max de la fusee retrouvé = 61.375078
justin@justin-linux:~/Bureau/TIPE$
```

Figure 11 - Résultats du programme Las Vegas

Maximiser le temps de vol :

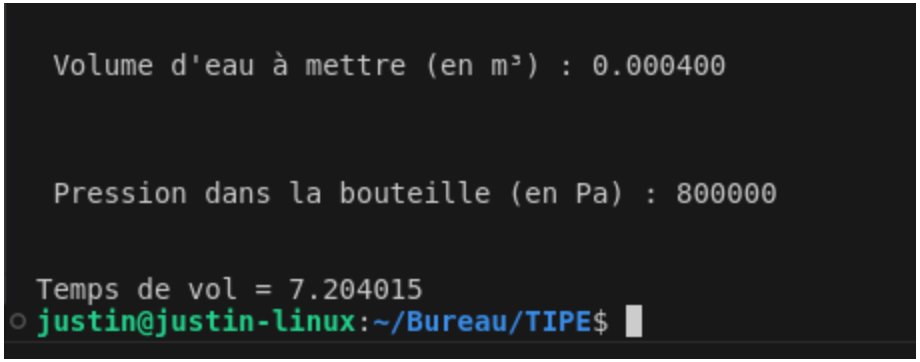


Figure 11 - Résultats du programme naïf

Précis à 0.01L d'eau près

Observation : On remarque que pression est toujours maximale, il reste à optimiser le volume d'eau

Validation du temps d'expulsion de l'eau

Figure 12 - Expulsion de l'eau

Vérification du temps de vol

	0.1	0.3	0.5	0.8	1
Résultat simulateur	3.41	4.18	4.03	1.41	/
Résultat expérience	3.03	3.77	3.68	1.23	≈0.3

Figure 13 - Résultats pour 3 bar

	0.1	0.3	0.5	0.8	1
Résultat simulateur	5.02	5.69	5.56	4.76	2.14
Résultat expérience	4.56	5.12	5.16	4.40	1.98

Figure 14 - Résultats pour 4 bar

Cf annexe pour un tableau récapitulatif et comparatif

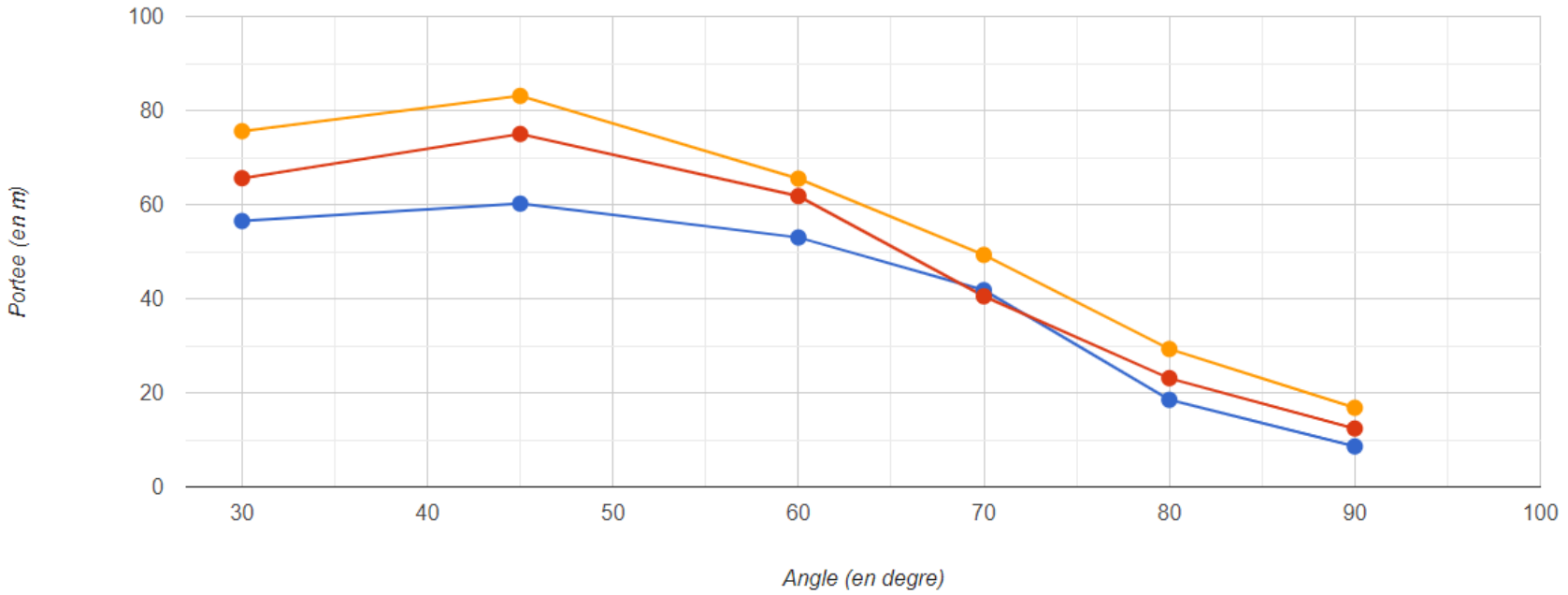


Figure 15 - Ordre d'idée de la portée

Remarque : Chaque point représente la moyenne de 2 lancés (les résultats obtenu étant différents , ce graphe ne donne qu'un ordre d'idée)

Distances d'atterrissage

Conclusion

- Le nombre de paramètres rentrant en jeu complexifient considérablement l'approche des phénomènes physiques

- Approximer les résultats des équations différentielles principales peut s'avérer relativement efficace au vu des faibles intervalles de temps

- Lors d'expérimentations réelles, les paramètres extérieurs influant le mouvement de la fusée rendent la simulation très peut précise

ANNEXE

Formule de Bernoulli utilisée :

Dans le cas d'un écoulement parfait (sans effet de viscosité ni perte de charges), pour un fluide incompressible, en considérant la variation d'énergie potentielle comme étant négligeable (faible intervalle de temps et faible masse) :

$$P - P_\infty = \frac{1}{2} \cdot \rho \cdot u^2 - \frac{1}{2} \cdot \rho \cdot u_\infty^2$$

Expression des données durant la phase de vol :

$$t_{top} = t_0 + \frac{v_t}{g} \cdot \arctan\left(\frac{v_0}{v_t}\right)$$

$$z(t) = \begin{cases} z_{top} + \dfrac{v_t^2}{g} \cdot \ln\left(\cos\left(\dfrac{g \cdot (t_{top} - t)}{v_t}\right)\right) & \text{monté} \\ z_{top} - \dfrac{v_t^2}{g} \cdot \ln\left(\cosh\left(\dfrac{g \cdot (t_{top} - t)}{v_t}\right)\right) & \text{descente} \end{cases}$$

$$z_{top} = z_0 + \frac{v_t^2}{2 \cdot g} \cdot \ln\left(1 + \left(\frac{v_0}{v_t}\right)^2\right)$$

$$t_{end} = t_{top} + \frac{v_t}{g} \cdot \operatorname{arctanh}\left(-\frac{v_{end}}{v_t}\right)$$

$$v_{end} = -v_t \cdot \sqrt{1 - \exp\left(-\frac{2 \cdot g \cdot z_{top}}{v_t^2}\right)}$$

Fonctions de calculs durant la phase de vol :

```c
#include "_fusee.h"

double  calcul_vt(fusee r, env e){
    double res;
    res = (double)2*(r.empty_mass)*e.g;
    double a = (e.rho)*(r.drag_coeff)*r.radius*_PI*r.radius;
    res = res/a;
    res = sqrt(res);
    return res;
}

// Renvoie le temps mis par la fusee pour atteintre le sommet de sa trajectoire
double calcul_t_top(double v_i, double v_t, env e, double t_i){
    double at = atan(v_i/v_t);
    double mul = v_t/e.g;
    return at*mul+t_i;
}

// correct à environ 1 m près d'après test, renvoie l'altitude max atteinte par la fusee
double calcul_z_top(double z_i,double v_t, double v_i, env e){
    double v = v_i/v_t;
    double l = log(1+(v*v));
    double v_t2 = (v_t*v_t);
    double mul = v_t2/(2*e.g);
    return z_i+mul*l;
}

// Renvoie la vitesse de la fusee a la fin de sa phase de propulsion
double calcul_v_i(fusee r, env e){
    double T[3] = {0.0,0.0,0.0};
    methode1(T,r,e);
    return T[1];
}
```

```
34
35 // Renvoie la vitesse de la fusee a la fin de son mouvement
36 double calcul_v_end(double vt, double z_max, env e){
37     double res = (double)2*e.g*z_max/pow(vt,2);
38     res = 0-res;
39     res = 1-exp(res);
40     if(res <= 0){res = 0;}
41     else{res = vt*sqrt(res);}
42     return 0-res;
43 }
44
45 // Renvoie le temps mis par la fusee pour effectuer son mouvement en entier
46 double calcul_t_end_fusee(fusee r, env e){
47     double T[3];
48     methode1(T,r,e);
49     double vt = calcul_vt(r,e);
50     double t_top = calcul_t_top(T[1],vt,e,T[0]);
51     double z_max = calcul_z_top_fusee(r,e);
52     double v_end = calcul_v_end(vt,z_max,e);
53     double res = (double) vt/e.g;
54     res = res*atanh(0-(v_end/vt));
55     return res+t_top;
56 }
```

Tentative en 2 dimensions :

```
68  //------------------------------------------------------------------------
69  // On essaye de prendre un angle de tire ce qui nous ramène à un problème en 2 dimensions
70  // On tente ici de considérer teta, étant un fonction du temps, comme une constante valant
71  // un peu moins que sa moitié (au vu de la trajectoire théorique de la fusee)
72  // Par rapport aux résultats d'expérimentations réelles, c'est un échec ...
73
74  double  calcul_vt_z(fusee r, env e, double teta){
75      double res;
76      res = (double)2*(r.empty_mass)*e.g;
77      double a = (e.rho)*(r.drag_coeff)*r.radius*_PI*r.radius*sin(teta);
78      res = res/a;
79      res = sqrt(res);
80      return res;
81  }
82
83  double  calcul_vt_x(fusee r, env e, double teta){
84      double res;
85      res = (double)2*(r.empty_mass)*e.g;
86      double a = (e.rho)*(r.drag_coeff)*r.radius*_PI*r.radius*cos(teta);
87      res = res/a;
88      res = sqrt(res);
89      return res;
90  }
91
92  // calcul_t_top reste la même fonction mais avec vt = vt_z ; de même pour v_end et t_end
93  double calcul_x_end(fusee r, env e, double teta){ // teta appartient à [0 ; pi/2]
94      double T[3];
95      methode1(T,r,e);
96      double vt_z = calcul_vt_z(r,e,teta*0.6);
97      double vt_x = calcul_vt_x(r,e,teta*0.6);
98
99      double t_ap = calcul_t_top(T[1], vt_z,e,T[0]);
100     double z_max = calcul_z_top(T[2], vt_z, T[1], e);
101     double v_end = calcul_v_end(vt_z, z_max, e);
102     double t_end = (double) vt_z/e.g;
103     t_end = t_end*atanh(0-(v_end/vt_z));
104     t_end += t_ap;
105
106     double v_x_0 = T[1]*cos(teta);
107     double terme1 = pow(v_x_0/vt_x,2); terme1 += 1;
108     terme1 = log(terme1); terme1 = terme1*(pow(vt_x,2)/(2*e.g));
109
110     double x_ap = v_x_0*T[0] + terme1;
111     double terme2 = (t_ap-t_end)/vt_x;
112     terme2 = cosh(terme2); terme2 = log(terme2);
113     terme2 = terme2*(pow(vt_x,2)/e.g);
114
115     return x_ap - terme2;
116 }
```

Fonctions de calculs durant la phase de poussée :

```
#include "_fusee.h"

//-------------------------------------------------------------------------
// Détermination des variables nécessaires aux calculs

double pourcent_eau(fusee r){
    return r.water_volume / r.empty_volume;
}

double temp2(fusee r, env e){
    double res;
    res = (double)1-pourcent_eau(r);
    res = pow(res,e.gamma-1);
    return res*e.T_ext;
}

double beta(env e){
    return 1.03+(0.021*e.gamma);
}

double press2(fusee r, env e){
    double res;
    res = (r.empty_volume - r.water_volume)/r.empty_volume;
    res = pow(res,e.gamma);
    return r.pressure*res;
}

double p_trans(env e){
    double res;
    res = (double)(e.gamma+1)/2;
    res = pow(res,e.gamma/(e.gamma-1));
    return e.P_ext*res;
}

double r_m = _R / 0.029;

double _c2(fusee r, env e){
    double t2 = temp2(r,e);
    double res;
    res = (double)e.gamma*t2*r_m;
    return sqrt(res);
}

```

```
//-------------------------------------------------------------------------
// Détermination des paramètres de fin de la phase de propulsion

double imp(fusee r, env e){ // Renvoie l'augmentation de la vitesse de la fusee après le "gas impulse"
    double p2 = press2(r,e);
    double b = beta(e);
    double pt = p_trans(e);
    double c2 = _c2(r,e);


    double terme1 = p2/(b*e.P_ext); terme1 = pow(terme1, (e.gamma-1)/(2*e.gamma)); terme1 = 1-terme1;
    terme1 = pt*terme1; terme1 = terme1 / (p2*(e.gamma-1));
    double terme2 = b*e.P_ext/p2; terme2 = pow(terme2, (e.gamma+1)/(2*e.gamma)); terme2 = 1-terme2;
    double terme3 = 8/(e.gamma+1); terme3 = sqrt(terme3);
    double terme4 = p2*r.empty_volume/c2;
    double res = (double) terme3*terme4;
    res = res * (terme1 + terme2);
    return res/r.empty_mass;
}

double _tau(fusee r, env e){
    double c2 = _c2(r,e);
    double a_star = r.nozzle_radius*_PI*r.nozzle_radius;
    double res = (double) r.empty_volume/(a_star*c2);
    res = res*(2/(e.gamma-1));
    double terme1 = (e.gamma+1)/2;
    terme1 = pow(terme1, ((e.gamma+1)/(2*(e.gamma-1))));
    res = res*terme1;
    return res;
}

double t_gas(fusee r,env e){    // Calcul du temps d'expulsion du gaz
    double tau = _tau(r,e);
    double p2 = press2(r,e);
    double b = beta(e);
    double term1 = p2/(b*e.P_ext);
    term1 = pow(term1, ((e.gamma-1)/(2*e.gamma)));
    double res = (double)term1-1;
    return tau*res;
}
```

1ère méthode d'approximation d'expulsion de l'eau :

```
85  // 1ère méthode pour expulsion de l'eau :
86  double v_e(fusee r, env e){ // Vitesse d'expulsion de l'eau dans le goulot
87      double p2 = press2(r,e);
88      double res = (double) 2*(p2 - e.P_ext);
89      res = res/_RHO_EAU;
90      return sqrt(res);
91  }
92
93  // Met a jour T avec les paramètres de la fin de l'expulsion de l'eau de la bouteille
94  void expulsion_eau_v1(double T[3], fusee r, env e){
95      double ve = v_e(r,e);
96      double a_star = r.nozzle_radius*_PI*r.nozzle_radius;
97      double a = r.radius*_PI*r.radius;
98      double debit = ve*a_star;
99
100     double t_f = (double) r.water_volume/debit;    // Calcul du temps après expulsion de l'eau
101     T[0] = t_f;
102
103     double m0 = r.empty_mass + (_RHO_EAU*r.water_volume);
104
105     double terme1 = (double)_RHO_EAU*a_star*ve*t_f; terme1 = terme1 / m0; terme1 = 1-terme1; // Calcul de la vitesse après expulsion de l'eau
106     double vf = (double) log(terme1);
107     vf = -ve*vf; vf = vf - e.g*t_f;
108     T[1] = vf;
109     double terme2 = log(terme1);  // Calcul de la position après expulsion de l'eau
110     double terme3 = m0/(_RHO_EAU*a);
111     double zf = (double)ve*t_f*(1-terme2);
112     zf += terme3*terme2;
113     zf -= 0.5*e.g*pow(t_f,2);
114     T[2] = zf;
115 }
116
117 void methode1(double T[3], fusee r, env e){ // T[0] = t_i ; T[1] = v_i ; T[2] = z_i
118     if (r.pressure != 0 && r.water_volume != 0)
119     {
120         expulsion_eau_v1(T, r, e);
121         double v = T[1];
122         T[1] += imp(r,e);
123         T[0] += t_gas(r,e);
124         double v_moy = (v+ T[1])/2;
125         T[2] += v_moy*t_gas(r,e);
126         //T contient les paramètres à la fin de la phase de propulsion
127     }
128 }
```

Tentative de lecture de coefficients polynomiaux dans un fichier :

Coefficients sous la forme :

a*x**9 - b*x**8 + c*x**7 - d*x**6 + e*x**5 - f*x**4 + g*x**3 - h*x**2 + i*x

```
131 // 2ème méthode pour expulsion de l'eau :
132 void extrait_coefficients(char* polynome,long  double coefficients[10]) {
133     for (int i = 0; i < 10; i+=1) {
134         coefficients[i] = 0.0;
135     }
136     long double a = 0, b = 0, c = 0, d = 0, e = 0, f = 0, g = 0, h = 0, i = 0, j = 0;
137 
138     sscanf(polynome, "%Lf*x**9 %Lf*x**8 %Lf*x**7 %Lf*x**6 %Lf*x**5 %Lf*x**4 %Lf*x**3 %Lf*x**2 %Lf*x %Lf", &a, &b, &c, &d, &e, &f, &g, &h, &i, &j);
139     coefficients[0] = a;
140     coefficients[1] = b;
141     coefficients[2] = c;
142     coefficients[3] = d;
143     coefficients[4] = e;
144     coefficients[5] = f;
145     coefficients[6] = g;
146     coefficients[7] = h;
147     coefficients[8] = i;
148     coefficients[9] = j;
149 }
```

```
150 void methode2_false(fusee r){   // NE MARCHE PAS (sûrement à cause du sscanf de la fonction précédente) : D'après tests, toutes les cases de coef sont à 0.0
151     FILE *file = fopen("resultats.txt", "r");
152     fseek(file, 0, SEEK_END);
153     long length = ftell(file);
154     fseek(file, 0, SEEK_SET);
155 
156     char *content = malloc(length + 1);
157 
158     fread(content, 1, length, file);
159     content[length] = '\0';
160     printf("\n%s\n", content);
161 
162     fclose(file);
163 
164     long double coef[10];
165     extrait_coefficients(content, coef);
166     free(content);
167 
168     double a_star = r.nozzle_radius*_PI*r.nozzle_radius;
169 
170     long double terme1 = (long double) _RHO_EAU*a_star;
171     terme1 = terme1/((2*r.empty_mass + _RHO_EAU*r.water_volume)/2);
172     long double v = 0.0;
173 
174     for (int i = 0; i < 10; i+=1)
175     {
176         v += coef[i]*pow(0.04,i);
177     }
178     printf("\n\n zvizbv %Lf\n\n", v);
179 }
```

Retrouver la fusée à partir de l'altitude :

```c
#include "_fusee.h"

// Fonctions pour retrouver les valeurs :

//---------------------------------------------------------------------
// Première "épreuve" de compétition

bool correct_fusee(fusee r, env e, double z_max){
    double z_top = calcul_z_top_fusee(r, e);
    if(z_top - 2 <= z_max && z_top + 2 >= z_max){return true;}
    return false;
}

void random_rocket(fusee* r){
    r->pressure = (rand() % 500)*1000 + 300000;
    r->water_volume = (rand() % 13) / 10000.0 + 0.0002;
}

fusee* z_top_inverse(env e, double z_max, double vo, double r, double r_n,double m){
    fusee* ro = (fusee*) malloc(sizeof(fusee));
    ro->drag_coeff = 0.5;
    ro->empty_mass = m;
    ro->empty_volume = vo;
    ro->nozzle_radius = r_n;
    ro->pressure = 0;
    ro->radius = r;
    ro->surface = 0;
    ro->water_volume = 0;

    srand(time(NULL));
     while (!correct_fusee(*ro,e, z_max)){
        random_rocket(ro);
    }
    printf("\nres = %lf\n", calcul_z_top_fusee(*ro,e));


    return ro;
}

fusee* retrouve_fusee_z(env e, double z_max, bouteille b){
    fusee* r = z_top_inverse(e, z_max, b.empty_volume, b.radius, b.nozzle_radius, b.empty_mass);
    return r;
}
```

Maximisation du temps de vol :

```
// Deuxieme "epreuve" de competition
fusee* temps_max_inverse(env e, double vo, double r, double r_n,double m){
    fusee* ro = (fusee*) malloc(sizeof(fusee));
    ro->drag_coeff = 0.5;
    ro->empty_mass = m;
    ro->empty_volume = vo;
    ro->nozzle_radius = r_n;
    ro->pressure = 0;
    ro->radius = r;
    ro->surface = ro->radius*_PI*ro->radius;
    ro->water_volume = 0;

    int i = 300, j = 2; // i correspond à 300000 Pa et j à 0.0002 m³ d'eau
    double temps = 0.0;
    fusee* new_ro = malloc(sizeof(fusee));

    new_ro->drag_coeff = 0.5;
    new_ro->empty_mass = m;
    new_ro->empty_volume = vo;
    new_ro->nozzle_radius = r_n;
    new_ro->pressure = 0;
    new_ro->radius = r;
    new_ro->surface = new_ro->radius*_PI*new_ro->radius;
    new_ro->water_volume = 0;

    while (i!=801){
        j = 20;
        while (j!=160){
            new_ro->pressure = (long)i*1000;
            new_ro->water_volume = (double) j/100000;

            if (calcul_v_i(*new_ro,e) > calcul_v_i(*ro,e))
            {
                ro->pressure = new_ro->pressure;
                ro->water_volume = new_ro->water_volume;
            }
            j+=1;
        }
        i+=1;
    }
    free(new_ro);
    temps = calcul_t_end_fusee(*ro,e);
    printf("\nres = %lf\n", temps);
    return ro;
}

fusee* fusee_max_t(env e, bouteille b){
    fusee* r = temps_max_inverse(e, b.empty_volume, b.radius, b.nozzle_radius, b.empty_mass);
    return r;
}
```

Header :
Bibliothèques / Macros

```c
#ifndef FUSEE
#define FUSEE

//-------------------------------------------------------------------------
// Bibliothèques utilisées (assert.h représentait un intérêt pour le débogage)
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <assert.h>
#include <math.h>   // Nécessite le rajout de -lm en option de compilation
#include <time.h>
#include <string.h>

//-------------------------------------------------------------------------
// Constantes utiles (macros)

#define _PI 3.14159
#define _R 8.315
#define _RHO_EAU 998

```

Structures

```
21 //---------------------------------------------------------------------
22 // Structures utilisées
23
24 struct rocket_t
25 {
26     double empty_volume;
27     double water_volume;
28     double radius;
29     double nozzle_radius;
30     double empty_mass;
31     double drag_coeff;
32     long pressure;
33     double surface;
34
35     // Volumes in m³ ; Mass in kg ; Radius in m ; Pressure in Pa
36
37 };
38 typedef struct rocket_t fusee;
39
40 struct bouteille_t
41 {
42     double empty_volume;
43     double radius;
44     double nozzle_radius;
45     double empty_mass;
46
47     // Volumes in m³ ; Mass in kg ; Radius in m
48
49 };
50 typedef struct bouteille_t bouteille;
51
52 struct environement_t
53 {
54     double g;
55     double rho;
56     double T_ext;
57     long P_ext;
58     double gamma;
59
60     // g in m.s^-2 ; rho in kg.m^-3 ; T_ext in K ; P_ext in Pa
61
62 };
63 typedef struct environement_t env;
```

Fonctions "inter fichiers"

```
65  //-------------------------------------------------------------------------
66  // Fonctions "inter fichiers"
67
68  double imp(fusee r, env e);
69
70  void expulsion_eau_v1(double T[3], fusee r, env e);
71
72  void methode1(double T[3], fusee r, env e);
73
74  double methode2(fusee r, env e);
75
76  double calcul_vt(fusee r, env e);
77
78  double calcul_v_i(fusee r, env e);
79
80  double calcul_t_top(double v_i, double v_t, env e, double t_i);
81
82  double calcul_z_top(double z_i,double v_t, double v_i, env e);
83
84  double calcul_z_top_fusee(fusee r, env e);
85
86  void moy_methode(double T[3], fusee r, env e);
87
88  fusee* retrouve_fusee_z(env e, double z_max, bouteille b);
89
90  double calcul_t_end_fusee(fusee r, env e);
91
92  fusee* fusee_max_t(env e, bouteille b);
93
94
95  #endif
```

Approximation de u (Python) :

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import interpolate
from scipy.interpolate import interp1d
from scipy import integrate
from scipy.integrate import solve_ivp
from sympy import *
from sympy.interactive import printing
printing.init_printing(use_latex=True)

k = 1.4  # Coefficient adiabatique
C = 78  # Constante liée aux conditions de la fusée
A = 0.00038  # Surface de la tuyère en m^2
rho = 1000  # Densité de l'eau en kg/m^3
Pa = 100000  # Pression atmosphérique en Pa

# Equation différentielle
def f(t, u):
    term = (0.5 * rho * u**2 + Pa) / C
    return - (k * C * A / rho) * term**((k + 1) / k)

# Fonction de résolution
def runge_kutta_4(f, u0, t0, tf, dt):
    t = np.arange(t0, tf, dt)
    u = np.zeros(len(t))
    u[0] = u0

    for i in range(1, len(t)):
        k1 = dt * f(t[i-1], u[i-1])
        k2 = dt * f(t[i-1] + 0.5*dt, u[i-1] + 0.5*k1)
        k3 = dt * f(t[i-1] + 0.5*dt, u[i-1] + 0.5*k2)
        k4 = dt * f(t[i-1] + dt, u[i-1] + k3)
        u[i] = u[i-1] + (k1 + 2*k2 + 2*k3 + k4) / 6

    return t, u

# Conditions initiales
u0 = 34.67  # = vitesse de flux que l'on concidère constante avec la méthode 1
t0 = 0
tf = 0.04
dt = 0.001
```

```python
# Résolution
t, u = runge_kutta_4(f, u0, t0, tf, dt)

f_interp = interp1d(t, u, kind='cubic')

# Approximation polynômiale
t_new = np.linspace(min(t), max(t), 1000)  # Points pour les polynômes de Lagrange
u_new = f_interp(t_new)

coefficients = np.polyfit(t, u, deg=4)
print(coefficients)
poly_func = np.poly1d(coefficients)
print("Fonction polynomiale approximée :")
print(poly_func + "\n\n\n")

x = symbols('x')
# Intégration du polynôme
F = integrate((coefficients[0]*x**4 + coefficients[1]*x**3 + coefficients[2]*x**2 + coefficients[3]*x + coefficients[4])**2, x)

print(F)

with open('resultats.txt', 'w') as fichier:
    fichier.write(str(F))


# Vérifier d'écriture
with open('resultats.txt', 'r') as fichier:
    contenu = fichier.read()
    print("\n\n\n" + contenu)


# Graphique
plt.plot(t_new, u_new, label='Interpolation')
plt.xlabel('Temps (s)')
plt.ylabel('Vitesse (m/s)')
plt.title('Interpolation de la fonction de la courbe')
plt.legend()
plt.grid(True)
plt.show()

# Résultats numériques
for i in range(len(t)):
    print(f"Temps: {t[i]:.2f} s, Vitesse: {u[i]:.2f} m/s")
```

Tracé de la vitesse durant la poussée (Python) :

```python
import math
import matplotlib.pyplot as plt
import numpy as np
from sympy import *
from sympy.interactive import printing


# Constants
_PI = math.pi
_RHO_EAU = 998.0  # Density of water in kg/m^3
_R = 8.314  # Universal gas constant in J/(mol*K)
_MOL_WEIGHT_AIR = 0.029  # Molar weight of air in kg/mol

# Class definitions
class Rocket:
    def __init__(self, empty_volume, water_volume, radius, nozzle_radius, empty_mass, drag_coeff, pressure, surface):
        self.empty_volume = empty_volume
        self.water_volume = water_volume
        self.radius = radius
        self.nozzle_radius = nozzle_radius
        self.empty_mass = empty_mass
        self.drag_coeff = drag_coeff
        self.pressure = pressure
        self.surface = surface

class Environment:
    def __init__(self, g, rho, T_ext, P_ext, gamma):
        self.g = g
        self.rho = rho
        self.T_ext = T_ext
        self.P_ext = P_ext
        self.gamma = gamma

```

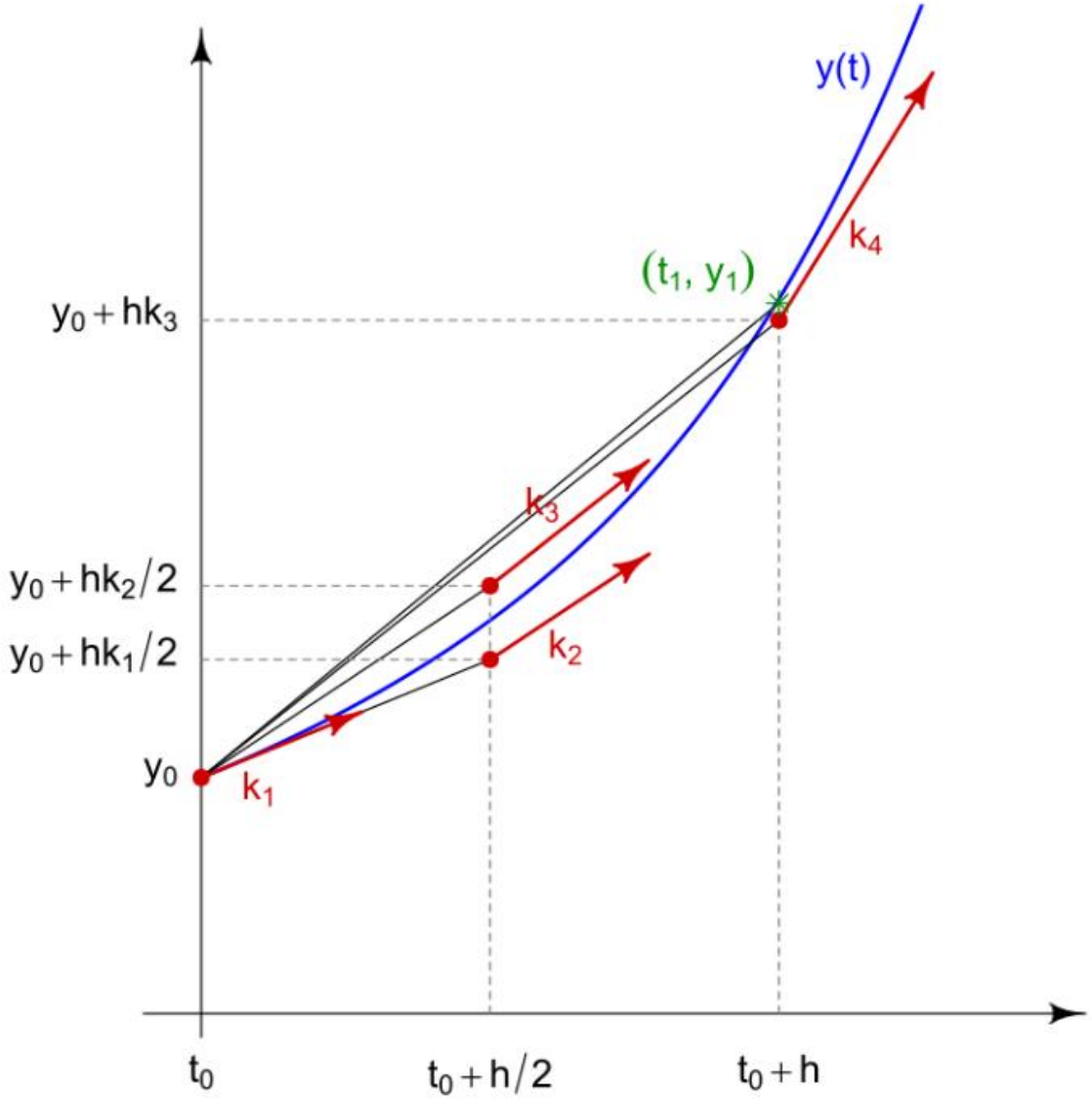
```python
def ft_gas(r, e, t, v):
    tau = _tau(r,e)
    a_star = r.nozzle_radius * _PI * r.nozzle_radius
    terme1 = 2 / (e.gamma + 1)
    terme1 = terme1**(1/(e.gamma - 1))
    terme1 = terme1*2*a_star*press2(r,e)
    f_t = 1 + (t/tau)
    f_t = f_t**(2*e.gamma/(-e.gamma + 1))
    f_t = f_t*terme1 - (e.P_ext*a_star)
    f_t = f_t / r.empty_mass
    f = f_t - e.g - 0.5*r.surface*e.rho*r.drag_coeff
    return f




def runge_kutta_4(r, e, f, u0, t0, tf, dt):
    t = np.arange(t0, tf, dt)
    u = np.zeros(len(t))
    u[0] = u0

    for i in range(1, len(t)):
        k1 = dt * f(r,e,t[i-1], u[i-1])
        k2 = dt * f(r,e,t[i-1] + 0.5*dt, u[i-1] + 0.5*k1)
        k3 = dt * f(r,e,t[i-1] + 0.5*dt, u[i-1] + 0.5*k2)
        k4 = dt * f(r,e,t[i-1] + dt, u[i-1] + k3)
        u[i] = u[i-1] + (k1 + 2*k2 + 2*k3 + k4) / 6

    return t, u

```

Les fonction _tau, … sont les même que celle du fichier de poussée en C

```python
def expulsion_eau_v1(T, r, e, time_intervals, velocities, dt):
    ve = v_e(r, e)
    a_star = r.nozzle_radius * _PI * r.nozzle_radius
    a = r.radius * _PI * r.radius

    debit = ve * a_star
    t_f = r.water_volume / debit
    T[0] = t_f

    m0 = r.empty_mass + (_RHO_EAU * r.water_volume)

    terme1 = _RHO_EAU * a_star * ve * t_f / m0
    terme1 = 1 - terme1

    vf = math.log(terme1)
    vf = -ve * vf - e.g * t_f
    T[1] = 0

    terme2 = math.log(terme1)
    terme3 = m0 / (_RHO_EAU * a)

    zf = ve * t_f * (1 - terme2)
    zf += terme3 * terme2
    zf -= 0.5 * e.g * math.pow(t_f, 2)
    T[2] = zf

    total_time = 0.0
    current_water_volume = r.water_volume

    while current_water_volume > 0:
        current_water_volume -= dt * ve * a_star
        if current_water_volume < 0:
            current_water_volume = 0
        r.water_volume = current_water_volume
        current_mass = r.empty_mass + _RHO_EAU * current_water_volume

        # Update velocity based on mass and thrust
        current_thrust = _RHO_EAU * a_star * ve * ve
        acceleration = current_thrust / current_mass - e.g
        T[1] += acceleration * dt

        time_intervals.append(total_time)
        velocities.append(T[1])
        total_time += dt
    T[1] = vf

```

```python
def methode1(T, r, e):
    if r.pressure != 0 and r.water_volume != 0:
        time_intervals = []
        velocities = []
        dt = 0.001  # Time step for recording

        expulsion_eau_v1(T, r, e, time_intervals, velocities, dt)

        t0 = 0.0
        tf = t_gas(r,e)
        dt = 0.001
        v = T[1]

        t_i, v_int = runge_kutta_4(r, e, ft_gas, v, t0, tf, dt)

        for i in range (len(t_i)) :
            t_i[i] += T[0]

        time_int = np.concatenate((time_intervals,t_i))
        velo = np.concatenate((velocities, v_int))

        T[1] += imp(r, e)
        T[0] += t_gas(r, e)
        v_moy = (v + T[1]) / 2.0
        T[2] += v_moy * t_gas(r, e)

        return time_int, velo
    return [], []

# Simulation and plotting
def simulate_rocket(r, e):
    T = [0.0, 0.0, 0.0]
    time_intervals, velocities = methode1(T, r, e)
    return time_intervals, velocities

# Example usage of the functions with some test data
r = Rocket(0.002, 0.0005, 0.05, 0.011, 0.1, 0.5, 700000, 0.0079)
e = Environment(9.81, 1.2, 293.15, 100000, 1.4)

time_intervals, velocities = simulate_rocket(r, e)


plt.plot(time_intervals, velocities)
plt.xlabel('Time (s)')
plt.ylabel('Velocity (m/s)')
plt.title('Vitesse lors de la poussee')
plt.grid(True)
plt.show()
```

Schéma de la méthode de Runge Kutta à l'ordre 4 :



Source : https://fr.wikipedia.org/wiki/Méthodes_de_Runge-Kutta

Tableau récapitulatif des résultats obtenus :

	Résultats des Expériences	**Résultats de mon Simulateur**	**Résultats du Simulateur en ligne**
Expulsion de l'eau	≈ 0.03 s	≈ 0.03 s	[0.02 s; 0.05 s]
Temps de vol (4 bar ; 0.5 L)	5.16 s	5.59 s	5.56 s
Altitude maximale (7 bar ; 0.5 L)	X	63.28 m	62.33 m

Bibliographie :

- A guide to building and understanding the physics of Water Rockets : https://www.npl.co.uk/skills-learning /outreach/water-rockets/wr_booklet_print.pdf

- Equations et calculs de trajectoires : http://www.et.byu.edu/~wheeler /benchtop/

- Analyse physique d'une fusée à eau : https://www.real-worldphysics-problems.com/water-rocket-physics.html

- Premier exemple de "compétition" : https://www.planete-sciences.org/espace/Rocketry-Challenge/Presentation#Missions-et-reglement-2024

- Deuxième exemple de "compétition" : https://www.simplyscience.ch/fr /jeunes/agenda/championnat-de-fusees-a-eau-jeunes#:~:text=But%20du%20championnat,'aide% 20d'une%20pompe

- Méthodes de Runge-Kutta : https://fr.wikipedia.org/wiki /Méthodes_de_Runge-Kutta

- Water Rocket Simulator : http://www. aircommandrockets.com/sim/simulator.htm