Pilotage automatique d'un voilier

Cabrillana Jean-Manuel

13 juin 2017

Plan

Problématique : Comment piloter un voilier de façon optimale ?

1. Calcul d'une route de navigation
   - Polaire des vitesses
   - Données météorologiques (fichier GRIB)
   - Algorithme de Dijkstra
2. Simulation du voilier
   - Le vent en mer
   - Forces hydrodynamiques
   - Dynamique du voilier
   - Commande par PID
3. Pilotage avec un réseau de Petri
   - Rôle et définition
   - Illustration du RdP
   - Optimisation du pilotage

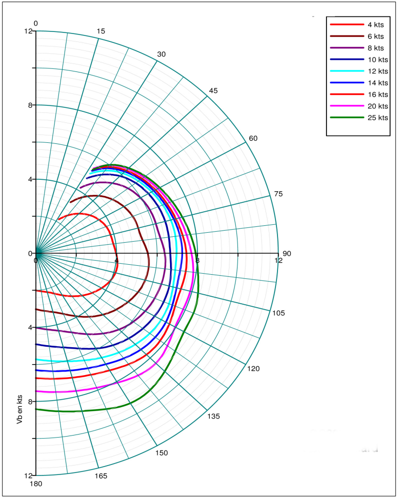


FIGURE – Vitesse du voilier en fonction de l'angle avec le vent et de sa vitesse

représentation cartographique

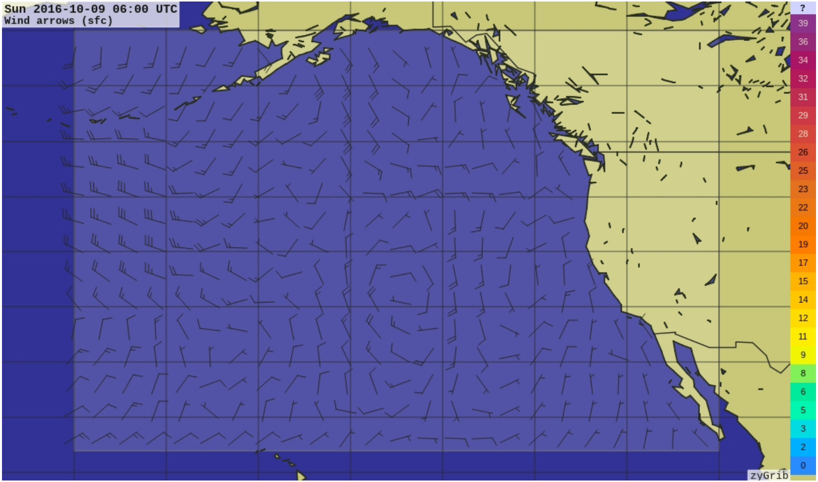
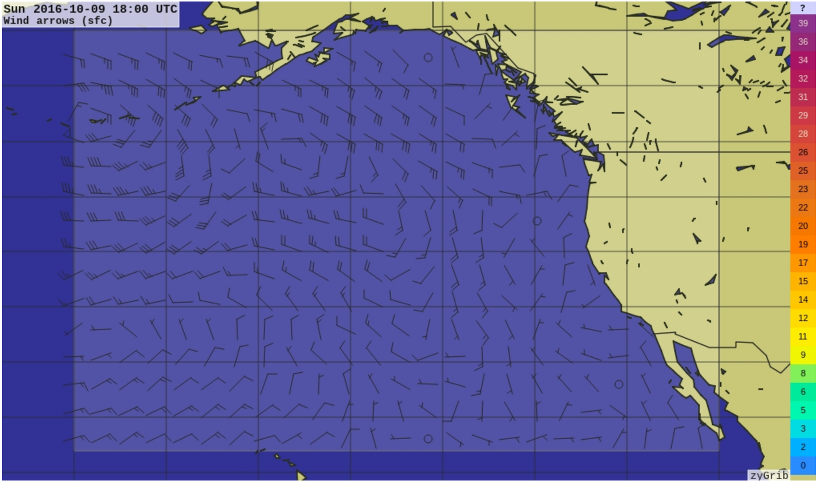
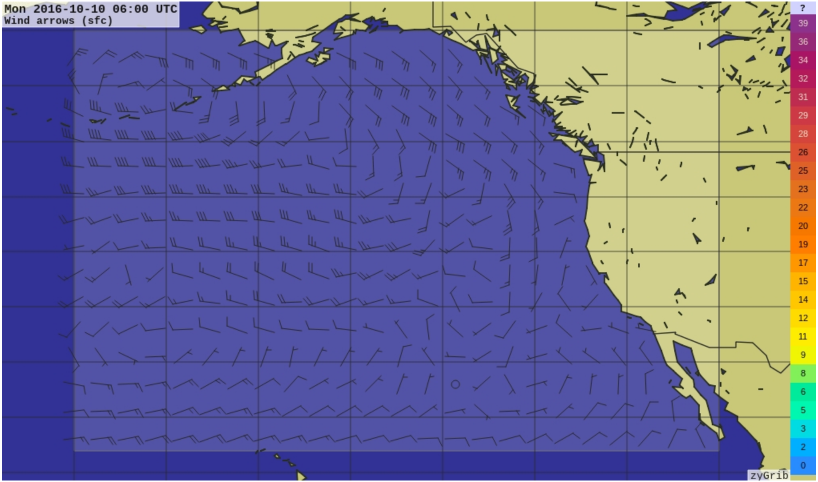


FIGURE – Golfe d'Alaska - Océan Pacifique

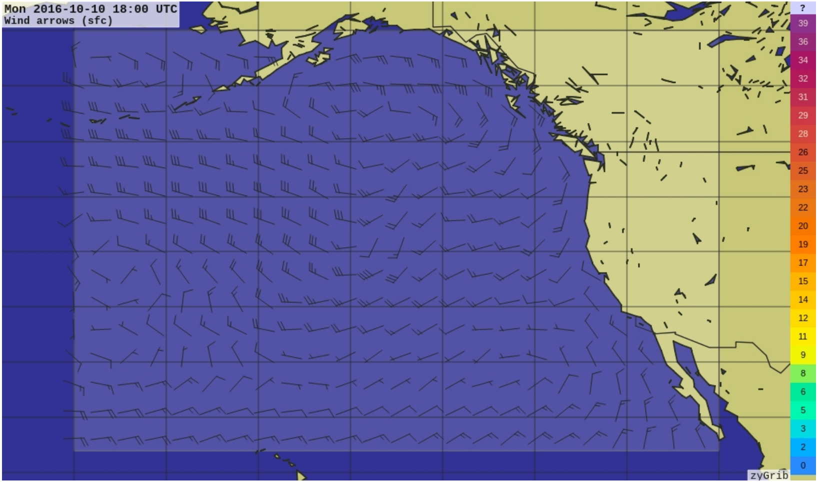
représentation cartographique



représentation cartographique



représentation cartographique



Structure de données

Structures utilisées :

- Un graphe G = (S, A)
- Les sommets de S sont des triplets (x,y,t) (aussi associés a un booléen "marqué" et un coût en temps)
- Une liste triée selon le coût au sommet de départ des voisins du sous-graphe (V)

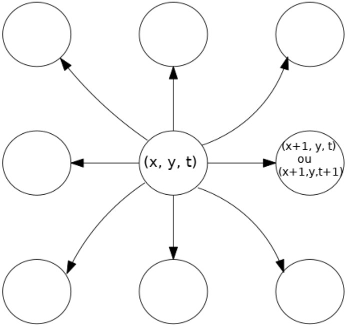


FIGURE – Exemple de sommets

Algorithme

Objectif

Construire un sous-graphe P, initialement vide, pour lequel les coût au sommet de départ sont minimaux ; renvoyer le chemin trouvé.

Algorithme

Tant qu'il existe un sommet dans V :

Choisir un sommet a dans V de plus petit coût

Mettre a dans P

Pour chaque sommet v hors de P voisin de a :

- v.cout $= \min(v.\text{cout}, a.\text{cout} + p(a, v))$
- Ajouter v à V si besoin

Fin pour et Fin Tant que

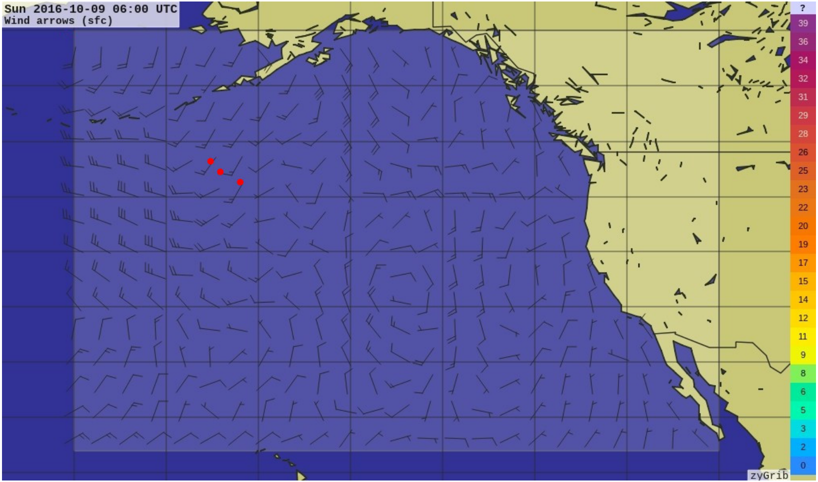
Retracer le chemin

Algorithme

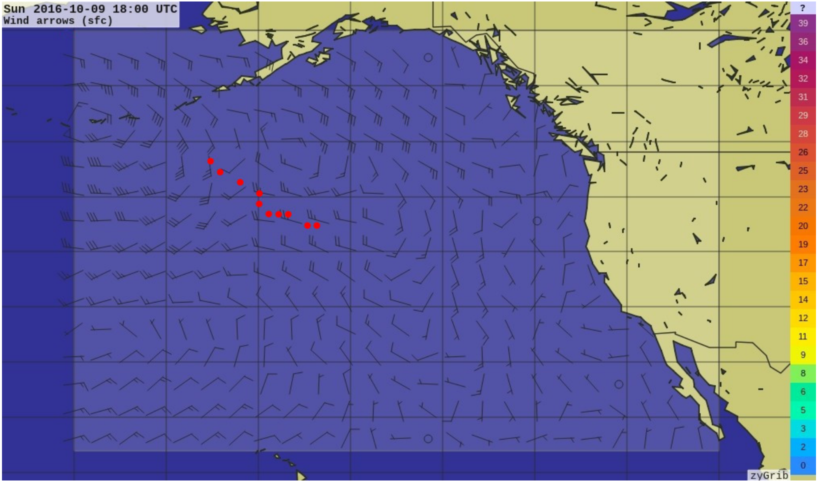
Complexité

La seule opération qui n'est pas à temps constant est la suppression et l'insertion dans V qui sont en temps logarithmique par rapport à sa taille, qui est au maximum de N. A chaque itération de l'algorithme, un sommet est traité. Donc il y a au plus N itérations. La complexité est en $\Theta(Nlog(N))$

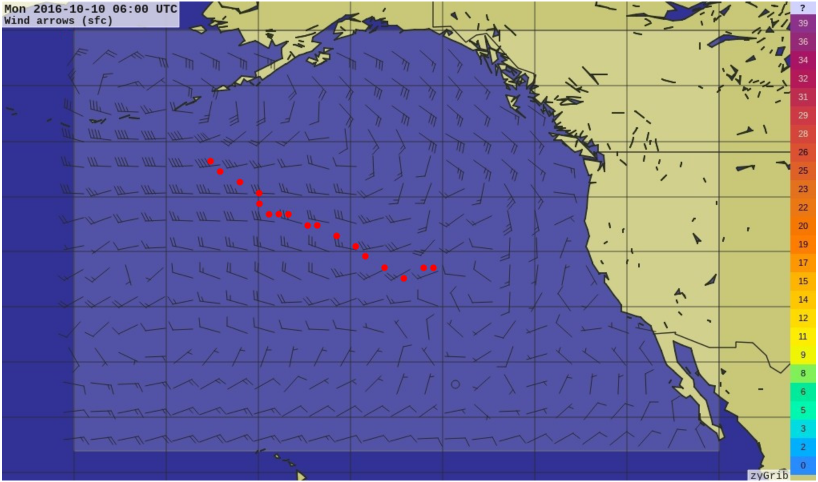
résultats


Sun 2016-10-09 06:00 UTC
Wind arrows (sfc)
?
39
36
34
32
31
29
28
26
25
23
22
20
19
17
15
14
12
11
9
8
6
5
3
2
0
zyGrib

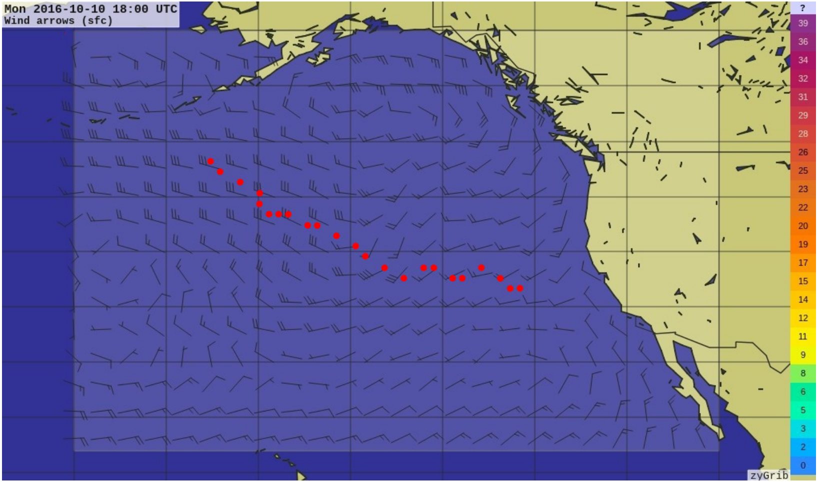
résultats


Sun 2016-10-09 18:00 UTC
Wind arrows (sfc)
?
39
36
34
32
31
29
28
26
25
23
22
20
19
17
15
14
12
11
9
8
6
5
3
2
0
zyGrib

résultats


Mon 2016-10-10 06:00 UTC
Wind arrows (sfc)
?
39
36
34
32
31
29
28
26
25
23
22
20
19
17
15
14
12
11
9
8
6
5
3
2
0
zyGrib

résultats


Mon 2016-10-10 18:00 UTC
Wind arrows (sfc)
?
39
36
34
32
31
29
28
26
25
23
22
20
19
17
15
14
12
11
9
8
6
5
3
2
0
zyGrib

II Simulation du voilier : du vent

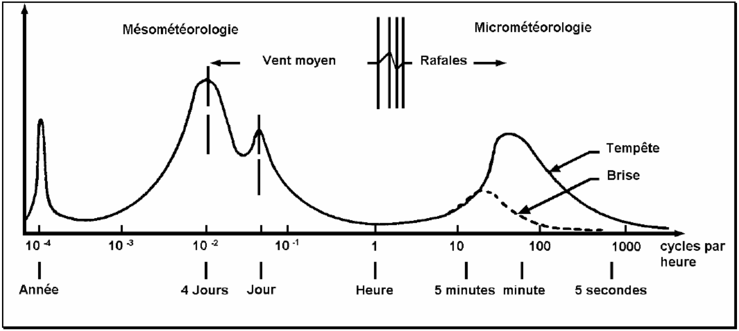


Figure 10.1 : Spectre de puissance du vent.

Forme discrétisée :

$$w(t) = \sum_{i=1}^{n} a_i cos(2\pi f_i + \phi_i)$$

Effort sur la voile

Norme de la force :

$$F_v = \frac{1}{2}\rho S V^2 C(\alpha)$$

Avec :

- ρ la masse volumique du fluide (l'air)
- S la surface de référence
- V la vitesse du vent
- C le coefficient aérodynamique
- α l'angle d'incidence avec le vent

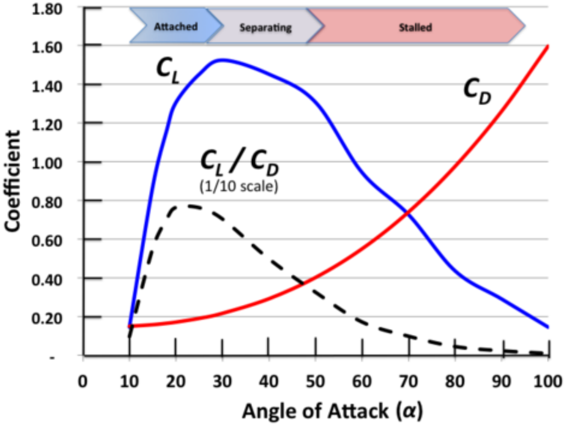


FIGURE – Coefficients de traînée et de portance de la voile

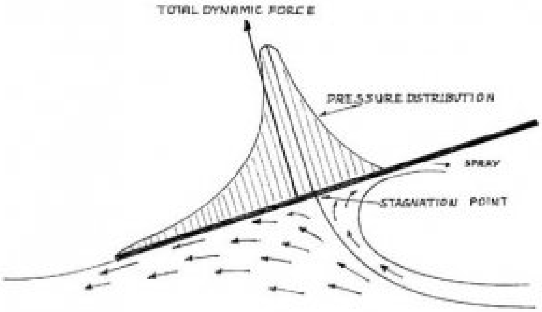
La voile est réglée au maximum de C_L/C_D Pour avoir le meilleur maintien de cap possible.

- Forces de trainé et de portance :

$$F = \frac{1}{2}\rho S V^2 C$$

- Force due aux vagues (force significative pour une vitesse de vent supérieure à 22 noeuds) :

$$F = K(\gamma) cos(wt - \vec{k}\,\mathbf{.}\,\vec{OM} + \phi)$$



dynamique du voilier

Théorème de la résultante cinétique au voilier :

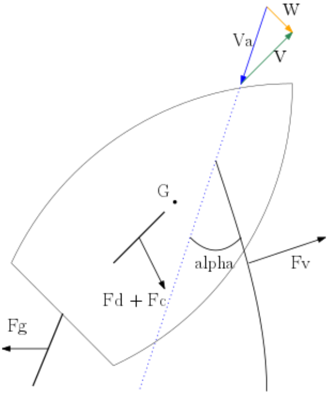
$$m\vec{a} = \sum \vec{F}$$

Théorème du moment cinétique selon l'axe (Gz) :

$$J\frac{d\omega}{dt} = \sum M$$

Equation de récurrence :

$$\vec{v_{i+1}} = \vec{v_i} + \frac{\sum \vec{F_i}}{m} dt$$



Régulateur PID agissant sur l'angle de barre β en fonction de l'écart avec le cap δ :

$$\beta = K(\delta + \frac{1}{\tau_d}\frac{d\delta}{dt} + \frac{1}{\tau_i}\int_0^t \delta(t)dt)$$

L'écart avec le cap peut être remplacé par l'écart avec le vent au près.

Methode du régleur

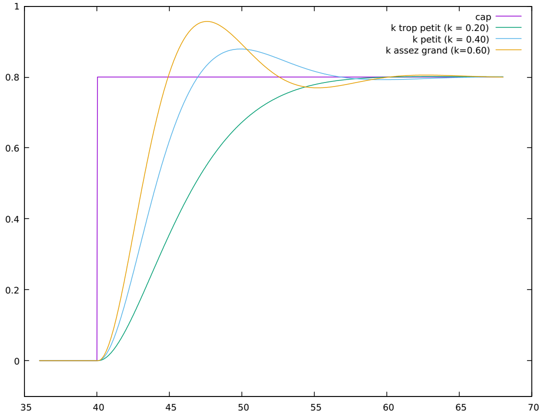


FIGURE – Réglage du gain K

Methode du régleur

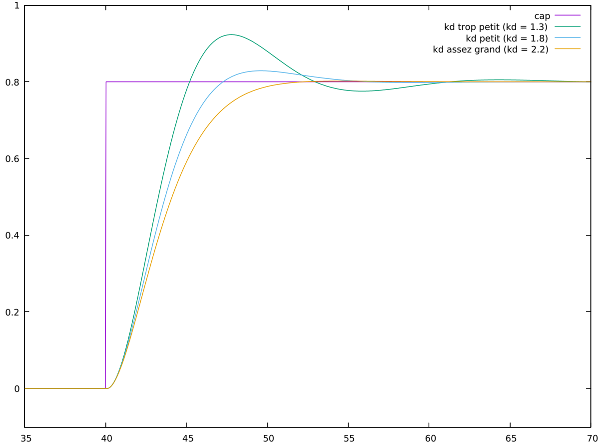
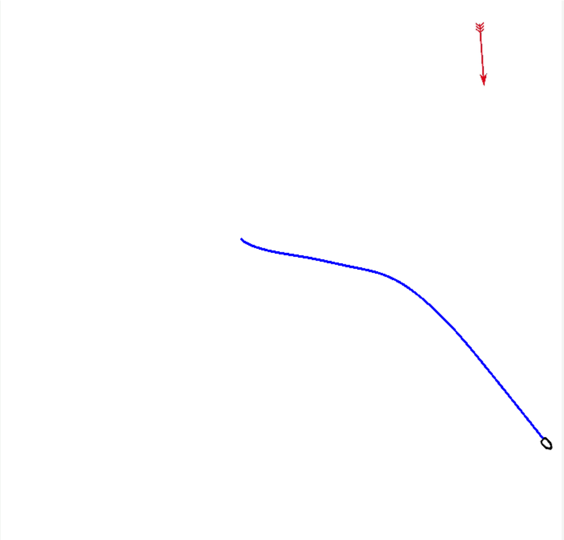


FIGURE – Réglage de Kd

Affichage



III Pilotage avec un réseau de Petri : Rôle et définition

Rôle

Le Réseau de Petri (RdP) est chargé de choisir la meilleure stratégie (relance, barre en mode cap ou vent) de pilotage en fonction des événements discrets qui pertubent le pilotage du voilier (rafale, molle, changement de direction du vent).

Définition

Un RdP est 5-tuplet $R = (P, T, post, pre, \tau)$ avec :

- P un ensemble non-vide fini de places
- $Pre : P \times T \rightarrow \mathbb{R}$ une application d'incidence avant
- $Post : P \times T \rightarrow \mathbb{R}$ une application d'incidence arriere
- $\tau \in \mathbb{N}^P$ est la temporisation associée aux places

Le marquage M d'un RdP est une application $M : P \rightarrow \mathbb{N}$

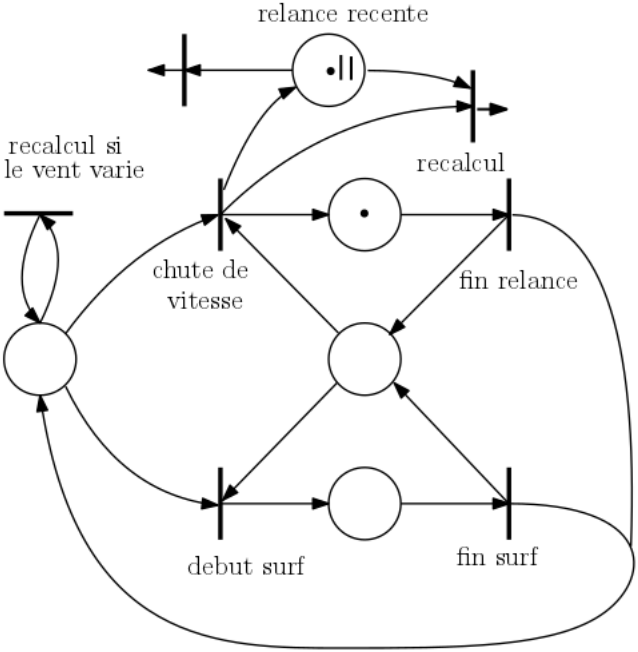


FIGURE – Illustration du RdP utilisé

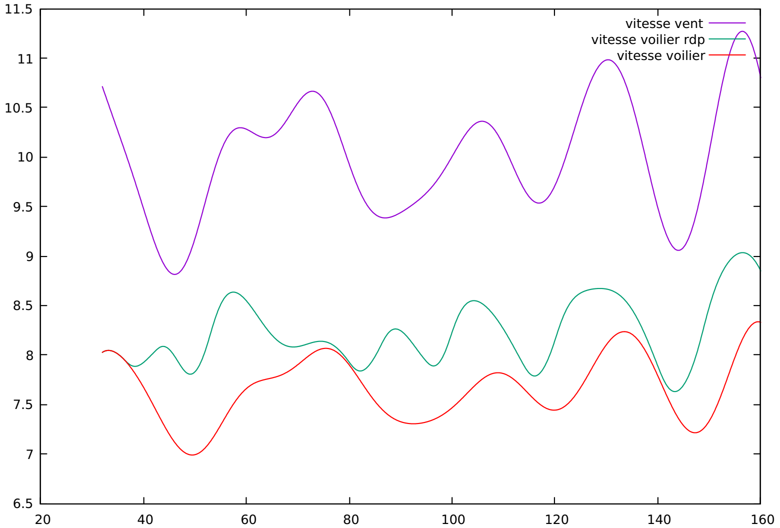


Figure – Comparaison des vitesses

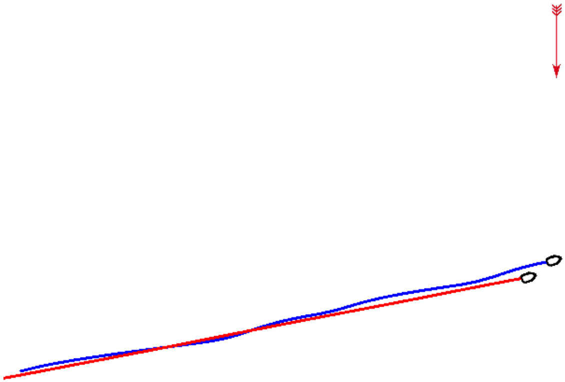


Figure – Trajectoire du voilier avec et sans RdP

Algorithme de Simulation

```
double spectre (double f)
{
    return (-300000*(f-0.003)*(f-0.06));
}

void coeff_spectre (double coeff[Nv])
{
  double df = 0.0020;
  double f = 0.003;
  for (int k = 0; k < Nv; k++)
  {
    coeff[k] = spectre(f)*df;
    f += df;
  }
}

double maj_vent (  double t)
{
    std::default_random_engine generator;
  std::normal_distribution<double> distribution (0, 0.5);
  double v = 10, f = 0.003, df = 0.0020;
  for (int k =0; k < Nv; k++)
  {
    double phi = distribution (generator);
    v += coeff[k]*cos(2*pi*f*t + phi);
    f += df;
  }
  return v;
}
```

```
vect force_voile (double beta, double delta, double kv, vect va)
{
    vect Fv(0, 0);
    std::vector <double> cl = {-3.420129, 32.100585, -84.563164, 117.32685, -91.27235,
    std::vector <double> cd = {0.130438, 0.085364, 0.097265,0.193015};
    if (abs(beta) < abs(delta) && abs(delta) >= 20*pi/180 && abs(beta) <= 85*pi/180)
    {
        Fv.y = kv*pow(norme(va), 2)*poly(cl, abs(abs(delta)-abs(beta)));
        Fv.x = kv*pow(norme(va), 2)*poly(cd, abs(abs(delta)-abs(beta)));
        if (beta >= 0)
        {
            Fv = rot(-(pi/2 - beta), Fv);
        }
        if (beta < 0)
        {
            Fv.y = -Fv.y;
            Fv = rot(pi/2 - beta, Fv);
        }
    }
    return Fv;
}
```

```
void maj_voilier (vect etat[5], const vect w)
{
    //x, y, vx, vy, theta, alpha, beta
    double theta = etat[3].x;
    double omega = etat[3].y;
    double alpha = etat[4].x;
    double beta = etat[4].y;
    double kg = 2, klat = 300, klong = 18, kv = 5;
    vect v = etat[1];
    vect vr = rot (-theta, v);
    vect a = etat[2];
    vect ez (0, 0, 1);
    vect er (cos(theta), sin(theta));
    vect ex (1, 0, 0);
    vect Gg (-2,0);
    vect var = rot(-theta, w - v);
    vect Fg (-cos(alpha)*kg*pow(vr.x,2), -sin(alpha)*kg*pow(vr.x,2));
    vect Fva ( -klong*pow(vr.x,2), -klat*abs(vr.y)*vr.y);
    double delta = vect_to_angle(-1*ex, var);
    vect Fv = force_voile (beta, delta, kv, var);
    a = 1/m*rot(theta, (Fg + Fv + Fva));
    vect dom =dt*v + 0.5*dt*dt*a;
    v = v + dt*a;
    etat[0] = etat[0] + dom;
    double domega = 1/J*((Gg^Fg)*ez - 0.000001*abs(omega)*omega);
    theta += dt*omega + 0.5*dt*dt*domega;
    omega += dt*domega;
    etat[1] = v;
    etat[2] = a;
    etat[3] = vect (theta, omega, domega);
}
```

```
void commande(vect etat[5], vect w, double cap)
{
    double k = 0.40 , kd = -2.0 ;
    double theta = etat[3].x;
    vect dir (cos(theta), sin(theta));
    vect dir_cap (cos(cap), sin(cap));
    double ecart = vect_to_angle (dir, dir_cap);

    double x = k*ecart + kd*etat[3].y ;
    if (not isnan(x)) etat[4].x = x;

    //reglage de beta
    vect v = etat[1];
    vect var = rot(-theta, w - v);
    vect ex (1, 0, 0);
    double delta = vect_to_angle(-1*ex, var);
    if (abs(delta) >= 27*pi/180 && abs(delta) < 60*pi/180)
        etat[4].y = delta - 22*pi/180*sgn(delta);
    if (abs(delta) >= 60*pi/180 )
        etat[4].y = delta - 27*pi/180*sgn(delta);
    if (abs(delta) >= 90*pi/180 )
        etat[4].y = delta - 32*pi/180*sgn(delta);
    if (abs(delta) >= 120*pi/180 )
        etat[4].y = 75;
    if (abs(delta) >= 150*pi/180 )
        etat[4].y = 85;
}
```

```
void simulation_cap (int n )
{
    //affichage fenêtre
    sf::Image grille;
    sf::Texture rendu;
    sf::Sprite affichage;
    grille.loadFromFile("grille.png");
    rendu.loadFromImage(grille);
    affichage.setTexture(rendu);
    window.clear();
    window.draw(affichage);
    window.display();
    sf::Texture img_voilier;
    img_voilier.loadFromFile("voilier.png");
    sf::Sprite voilier;
    voilier.setTexture(img_voilier);
    sf::Texture img_fleche;
    img_fleche.loadFromFile("fleche.jpg");
    sf::Sprite fleche;
    fleche.setTexture(img_fleche);
    voilier.scale(0.1, 0.1);
    voilier.setOrigin(voilier.getLocalBounds().width /2, voilier.
    fleche.setOrigin(fleche.getLocalBounds().width /2, fleche.get
    fleche.scale(0.2, 0.2);

    int tx=grille.getSize().x;
    int ty=grille.getSize().y;
    fleche.setPosition(tx-100,70);
    double cap = -0.3, phi = 90*pi/180, phi_p = 90*pi/180;
    double t = 0;
    //vecteur (r, v, a, theta, (alpha, beta))
    vect etat [5];
    etat[0].x = 300;
    etat[0].y = 300;
    etat[1].x = 0;
```

```
//angle aléatoire
std::default_random_engine generator;
std::normal_distribution<double> distribution (phi, 0.5);

for (int k = 0; k < n; k++)
{
if (k % 500 == 0) phi_p = distribution(generator);
phi += (phi_p - phi)/100;
double nw = maj_vent(t);
w.x = nw*cos(phi);
w.y = nw*sin(phi);
maj_voilier (etat, w);
commande (etat, w, cap);
t += dt;
int x = floorf(etat[0].x);
int y = floorf(etat[0].y);
if (etat[0].x > tx - 5) {etat[0].x -= tx-5; grille.loadFromFile("grille.png");}
if (etat[0].y > ty - 5) {etat[0].y -= ty-5; grille.loadFromFile("grille.png");}
if (etat[0].x <  5) {etat[0].x += tx-5; grille.loadFromFile("grille.png");}
if (etat[0].y <  5) {etat[0].y += ty-5; grille.loadFromFile("grille.png");}

    //affichage
    if ( x>5 && y > 5 && x <tx-5 && y< ty-5)
    {
        grille.setPixel(x, y, sf::Color::Blue);
        grille.setPixel(x+1, y, sf::Color::Blue);
        grille.setPixel(x-1, y, sf::Color::Blue);
        grille.setPixel(x, y+1, sf::Color::Blue);
        grille.setPixel(x, y-1, sf::Color::Blue);
        voilier.setPosition(x,y);
        voilier.setRotation(etat[3].x*180/pi + 90);
        fleche.setRotation(phi*180/pi);

        rendu.update(grille);
        affichage.setTexture(rendu);
        window.clear();
```

Algorithme de Dijkstra

```
const double infini = 10000;
const int larg = 15;
const int temps_max = 27;
const int pas_temps = 6;
const int pas_x = 1;
const int pas_y = 1;
const int Nx = 71;
const int Ny = 39;

double square (double x){return x*x;};

struct noeud {
    int i,j, temps;
    double cout;
    bool marque;
    noeud(): i(0), j(0), marque(false) {};
    noeud(int x, int y, int t ): i(x), j(y), temps(t), marque(false){};
    struct noeud * parent = NULL;
    bool operator== (const noeud& a ){return (i == a.i && j == a.j && temps == a.temps); };
};

struct comp_vent{
    double u,v;
    bool obs;
    comp_vent(): u(0), v(0), obs(false) {};
    comp_vent(double x, double y, bool z): u(x), v(y), obs(z) {};
};

struct noeud_compare{
    bool operator()(const noeud& a, const noeud& b){return a.cout < b.cout;};
};
```

```
double cout_arc (noeud co, noeud vo)
{
    int t = co.temps;
    double u = 0.5 * (grille[co.i][co.j][t].u + grille[vo.i][vo.j][t].u);
    double v = 0.5 * (grille[co.i][co.j][t].v + grille[vo.i][vo.j][t].v);
    double a = vect_to_angle (double (vo.i - co.i), double(vo.j - co.j));
    double b = vect_to_angle (-u, -v);
    //pour ramener l'angle dans [0,pi]
    double theta;
    if (abs(b-a) > 3.14) theta = abs(b-a) - 3.14;
    else theta = abs(b-a);
    double vi =   0.04 * sqrt(0.1 + (u*u + v*v))*poly(theta);
    double cout = sqrt (square(pas_x * (vo.i -co.i)) + square( pas_x * (vo.j -co.j)))/vi;
    return std::min(infini , cout);
}

void pop_word (std::string &str)
{
    if (not str.empty())
    {
        int k = 0;
        while (str[k] != ' ' ) k++;
        while (str[k] == ' ') k++;
        str = str.substr(k, str.length() - 1);
    }
}
```

```
void bordures (comp_vent grille[Nx][Ny][temps_max])
{
        for (int i=0; i < Nx; i++)
        {
            for (int t=0; t < temps_max; t++)
          {
                grille[i][0][t].obs = true;
                grille[i][Ny-1][t].obs = true;
                grille[i][1][t].obs = true;
                grille[i][Ny-2][t].obs = true;
          }
        }
        for (int j=0; j < Ny; j++)
        {
            for (int t=0; t < temps_max; t++)
          {
                grille[0][j][t].obs = true;
                grille[Nx-1][j][t].obs = true;
                grille[1][j][t].obs = true;
                grille[Nx-2][j][t].obs = true;
          }
        }
}


void initialise (noeud G[Nx][Ny][temps_max], noeud deb)
{
        for (int i=0; i < Nx; i++)
          {for (int j=0; j < Ny; j++)
            {for (int t=0; t < temps_max; t++)
              {
                (G[i][j][t]).cout = infini;
              }
            }
          }
        G[deb.i][deb.j][deb.temps].cout = 0;
}
```

```
typedef std::multiset<noeud, noeud_compare>::iterator It;

void supprimer (std::multiset<noeud, noeud_compare> &V, noeud v)
{
    std::pair<It, It> range = V.equal_range(v);
    It k = range.first;
    while ( k != range.second){
        if (v == (*k))
            V.erase (k++);
        else k++;}
}
```

```
void dijkstra ( noeud deb, noeud fin)
{
    initialise (G, deb);
    // liste triée selon la distance à deb des voisins du sous-graphe minimal
    std::multiset <noeud, noeud_compare> V;
    V.insert(deb);
    noeud courant;
    while (not V.empty() and not (courant.i == fin.i and courant.j == fin.j))
    {
        courant = *(V.begin());
        V.erase(V.begin());
        G[courant.i][courant.j][courant.temps].marque = true;
        //mise a jour de V pour les voisins de courant
        for ( int i = -2; i <= 2; i++)
        {
            for (int j = -2; j <= 2; j++)
            {
                if ((i != 0 or j != 0) and (abs (i) != 2 or j != 0) and (i != 0 or abs (j) != 0) and (abs (i) != 2 or al
                {
                    noeud v;
                    v.i=courant.i + i;
                    v.j=courant.j + j;
                    //on calcule le cout s'il n'y a pas d'obstacle et sinon rien n'est a mettre à jour
                    if (not grille[v.i][v.j][courant.temps].obs )
                    {
                        v.cout = courant.cout + cout_arc( courant, v);
                        v.temps = int (courant.cout) / pas_temps;
                        std :: cout << v.i << ' ' << v.j << ' ' << v.temps<< ' ' << cout_arc ( courant,v) << '\n';
                        //le sommet est mis à jour si il n'est pas marqué, et le cout est plus bas
                        if ( G[v.i][v.j][v.temps].marque == false && v.cout < G[v.i][v.j][v.temps].cout)
                        {
                             std:: cout << "ajouté!"<< '\n';
                             v.parent = &(G[courant.i][courant.j][courant.temps]);
                             G[v.i][v.j][v.temps] = v;
                             supprimer(V, G[v.i][v.j][v.temps]);
                             V.insert(v);
                        }
```