TOULOUSE INP N7

Problème de navigation, une approche MPC

- Date : 2025
- Durée approximative : inconnue

On considère un navire dans un courant constant $w = (w_x, w_y)$, $\|w\| < 1$. L'angle de cap) est contrôlé, amenant aux équations différentielles suivantes :

$$\begin{aligned} \dot{x}(t) &= w_x + \cos\theta(t), \quad t \in [0, t_f] \\ \dot{y}(t) &= w_y + \sin\theta(t), \\ \dot{\theta}(t) &= u(t). \end{aligned}$$

La vitesse angulaire est limitée et normalisée : $\|u(t)\| \leq 1$. Il y a des conditions aux limites au temps initial $t = 0$ et au temps final $t = t_f$, sur la position (x, y) et sur l'angle θ. L'objectif est de minimiser le temps final. Ce sujet est inspiré de ce TP dont le problème vient d'une collaboration entre l'Université Côte d'Azur et l'entreprise française CGG qui s'intéresse aux manoeuvres optimales de très gros navires pour la prospection marine.

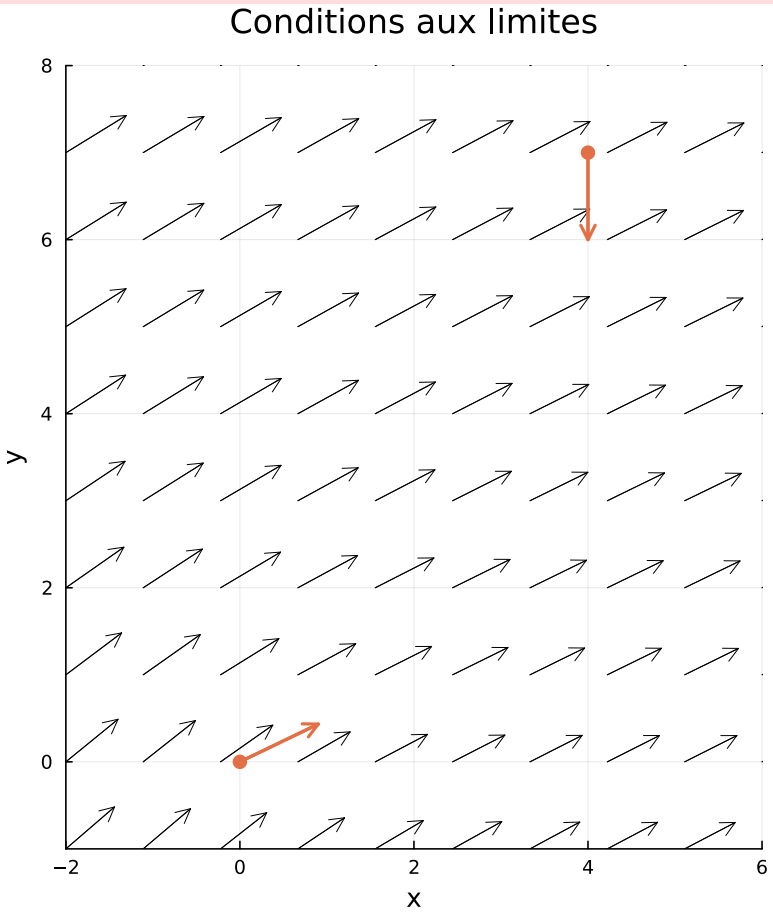
✏️ **Exercice 1.** On supposera $p^0 = -1$, on se place dans le cas normal.

1. Ecrire le problème de contrôle optimal sous la forme de Mayer.
2. Donner le pseudo-hamiltonien $H(q, p, u)$, où $q = (x, y, \theta)$ et $p = (p_x, p_y, p_\theta)$.
3. Calculer l'équation adjointe, c'est-à-dire vérifiée par le vecteur adjoint p, donnée par le principe du maximum de Pontryagin.
4. Calculer le contrôle maximisant en fonction de p_θ (on pourra l'écrire comme une fonction multivaluée).
5. Calculer le contrôle singulier, c'est-à-dire celui permettant de vérifier $p_\theta(t) = 0$ sur un intervalle de temps non réduit à un singleton.

Données du problème

In [1]:
```
using OptimalControl, NLPModelsIpopt, Plots, OrdinaryDiffEq, LinearAlgebra, Plots.PlotMeasures

t0 = 0.
x0 = 0.
y0 = 0.
θ0 = π/7
xf = 4.
yf = 7.
θf = -π/2

function current(x, y) # current as a function of position
    ε = 1e-1
    w = [ 0.6, 0.4 ]
    δw = ε * [ y, -x ] / sqrt(1+x^2+y^2)
    w = w + δw
    if (w[1]^2 + w[2]^2 >= 1)
        error("|w| >= 1")
    end
    return w
end

#
function plot_state!(plt, x, y, θ; color=1)
    plot!(plt, [x], [y], marker=:circle, legend=false, color=color, markerstrokecolor=color, markersize=5, z_o
    quiver!(plt, [x], [y], quiver=([cos(θ)], [sin(θ)]), color=color, linewidth=2, z_order=:front)
    return plt
end
```

```
function plot_current!(plt; current=current, N=10, scaling=1)
    for x ∈ range(xlims(plt)..., N)
        for y ∈ range(ylims(plt)..., N)
            w = scaling*current(x, y)
            quiver!(plt, [x], [y], quiver=([w[1]], [w[2]]), color=:black, linewidth=0.5, z_order=:back)
        end
    end
    return plt
end

# on affiche dans le plan de phase augmenté les conditions aux limites et le courant
plt = plot(
    xlims=(-2, 6),
    ylims=(-1, 8),
    size=(600, 600),
    aspect_ratio=1,
    xlabel="x",
    ylabel="y",
    title="Conditions aux limites",
    leftmargin=5mm,
    bottommargin=5mm,
)

plot_state!(plt, x0, y0, θ0; color=2)
plot_state!(plt, xf, yf, θf; color=2)
plot_current!(plt)
```



In [2]:

```
function plot_trajectory!(plt, t, x, y, θ; N=5) # N : nombre de points où l'on va afficher θ

    # trajectoire
    plot!(plt, x.(t), y.(t), legend=false, color=1, linewidth=2, z_order=:front)

    if N > 0

        # longueur du trajet
        s = 0
        for i ∈ 2:length(t)
            s += norm([x(t[i]), y(t[i])] - [x(t[i-1]), y(t[i-1])])
        end

        # intervalle de longueur
        Δs = s/(N+1)
        tis = []
        s = 0
        for i ∈ 2:length(t)
```

```
            s += norm([x(t[i]), y(t[i])] - [x(t[i-1]), y(t[i-1])])
            if s > Δs && length(tis) < N
                push!(tis, t[i])
                s = 0
            end
        end

        # affichage des points intermédiaires
        for ti ∈ tis
            plot_state!(plt, x(ti), y(ti), θ(ti); color=1)
        end

    end

    return plt

end;
```

Solveur (OptimalControl)

✏️ **Exercice 2.** Coder le problème de contrôle optimal ci-dessous.

On pourra s'insipirer du problème de la rame de métro à temps minimal décrit dans la documentation du package OptimalControl pour définir notre problème de manoeuvre de navire ci-après.

In [3]:
```
function solve(t0, x0, y0, θ0, xf, yf, θf, w;
    grid_size=300, tol=1e-8, max_iter=500, print_level=4, display=true)

    # ----------------------------------------
    # Définition du problème : TO UPDATE
    ocp = @def begin

        tf ∈ R, variable
        t ∈ [t0, tf], time
        q = (x, y, θ) ∈ R³, state
        u ∈ R, control

        -1 ≤ u(t) ≤ 1

        q(t0) == [ x0, y0, θ0 ]
        q(tf) == [ xf, yf, θf ]

        q̇(t) == [ w[1]+cos(θ(t)),
                  w[2]+sin(θ(t)),
                  u(t)]

        tf → min

    end
    # ----------------------------------------

    # Initialisation
    tf_init = 1.5*norm([xf, yf]-[x0, y0])
    x_init(t) = [ x0, y0, θ0 ] * (tf_init-t)/(tf_init-t0) + [xf, yf, θf] * (t-t0)/(tf_init-t0)
    u_init = (θf - θ0) / (tf_init-t0)
    init = (state=x_init, control=u_init, variable=tf_init)

    # Résolution
    sol = OptimalControl.solve(ocp;
        init=init,
        grid_size=grid_size,
        tol=tol,
        max_iter=max_iter,
        print_level=print_level,
        display=display,
        disc_method=:euler,
    )

    # Récupération des données utiles
    t = time_grid(sol)
    q = state(sol)
    x = t -> q(t)[1]
    y = t -> q(t)[2]
    θ = t -> q(t)[3]
    u = control(sol)
    tf = variable(sol)

    return t, x, y, θ, u, tf, iterations(sol), sol.solver_infos.constraints_violation
```

```
end;
```

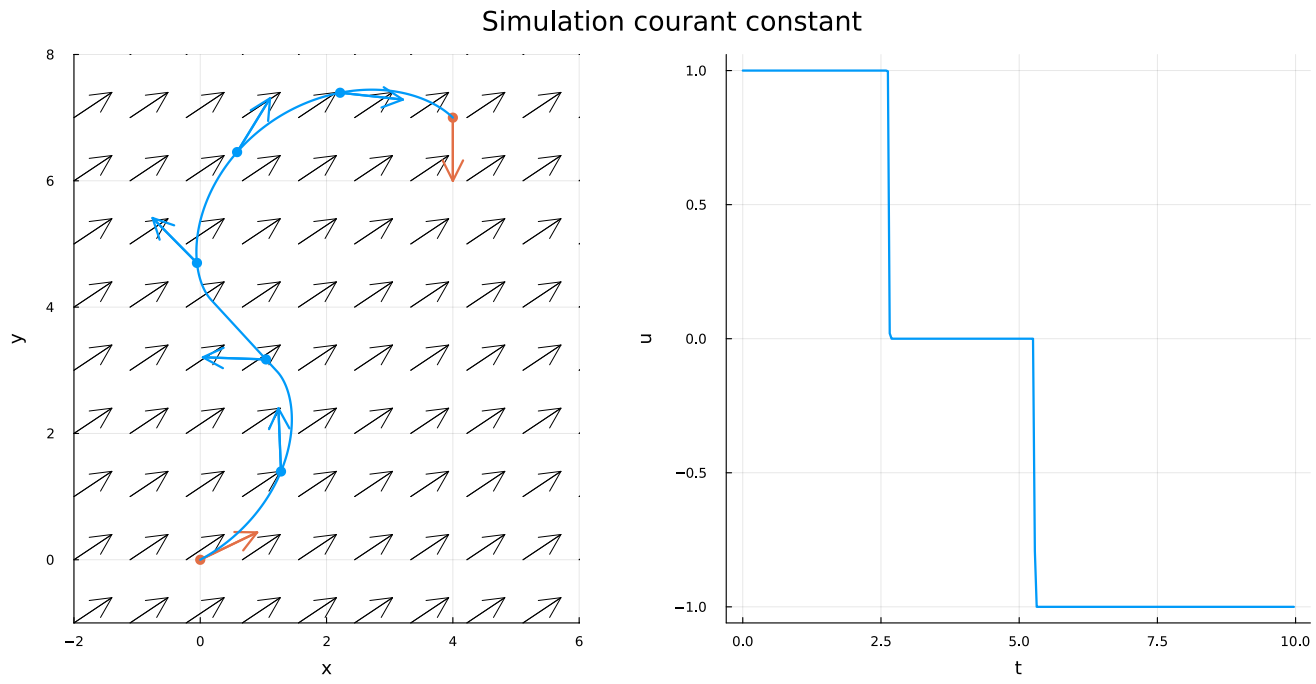
Première résolution

On considère ici un vent constant et on résout une première fois le problème.

In [4]:
```
# ------------------------------------------------------------------------------------------------------------------------
t, x, y, θ, u, tf, iter, cons = solve(t0, x0, y0, θ0, xf, yf, θf, current(x0, y0); display=false);

println("Iterations : ", iter)
println("Constraints violation : ", cons)
println("tf : ", tf)

# ------------------------------------------------------------------------------------------------------------------------
# affichage

# trajectoire
plt_q = plot(xlims=(-2, 6), ylims=(-1, 8), aspect_ratio=1, xlabel="x", ylabel="y")
plot_state!(plt_q, x0, y0, θ0; color=2)
plot_state!(plt_q, xf, yf, θf; color=2)
plot_current!(plt_q; current=(x, y) -> current(x0, y0))
plot_trajectory!(plt_q, t, x, y, θ)

# contrôle
plt_u = plot(t, u; color=1, legend=false, linewidth=2, xlabel="t", ylabel="u")

#
plot(plt_q, plt_u;
    layout=(1, 2),
    size=(1200, 600),
    leftmargin=5mm,
    bottommargin=5mm,
    plot_title="Simulation courant constant"
)
```

```
Iterations : 42
Constraints violation : 9.122780308956635e-10
tf : 9.969251026991449
```

Out[4]:



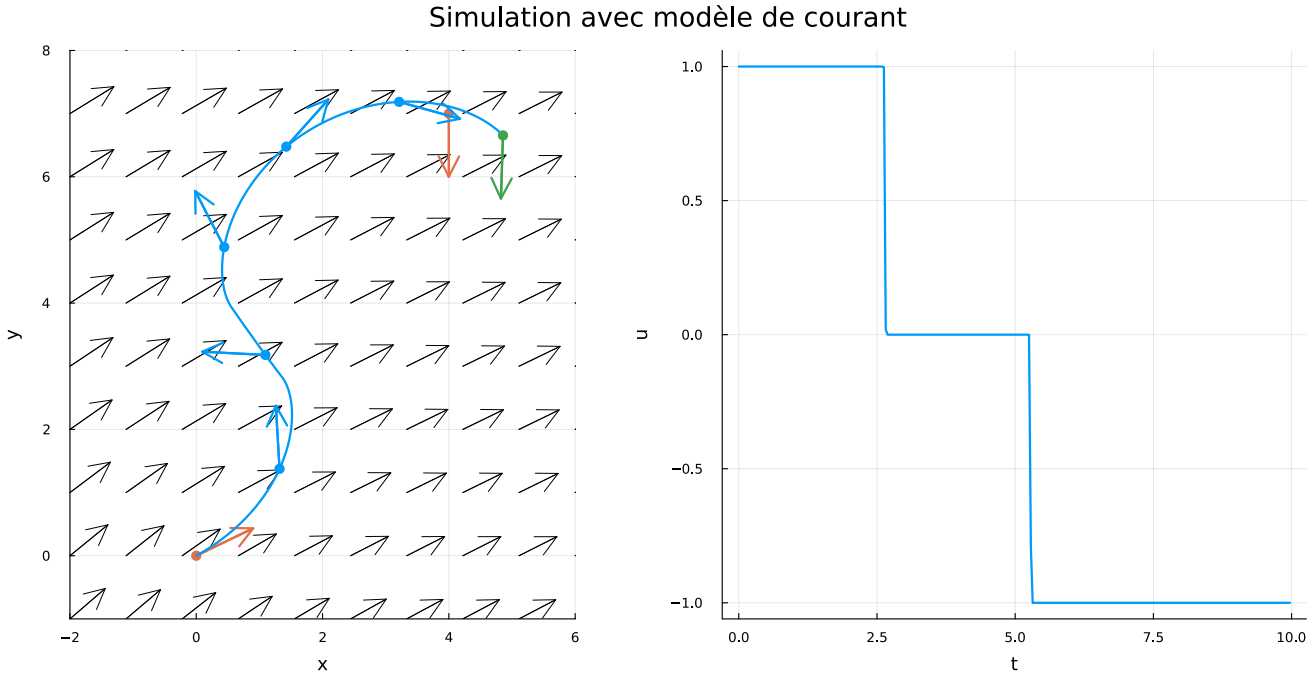
Simulation du système réel

Dans la simulation précédente, nous faisons l'hypothèse que le courant est constant. Cependant, d'un point de vue pratique le courant dépend de la position (x, y). Etant donné un modèle de courant, donné par la fonction `current`, nous pouvons simuler la trajectoire réelle du navire, pourvu que l'on ait la condition initiale et le contrôle au cours du temps.

In [5]:
```
function realistic_trajectory(tf, t0, x0, y0, θ0, u, current; abstol=1e-12, reltol=1e-12, saveat=[])

    function rhs!(dq, q, dummy, t)
        x, y, θ = q
        w = current(x, y)
        dq[1] = w[1]+cos(θ)
        dq[2] = w[2]+sin(θ)
        dq[3] = u(t)
    end

    q0 = [ x0, y0, θ0 ]
    tspan = (t0, tf)
    ode = ODEProblem(rhs!, q0, tspan)
    sol = OrdinaryDiffEq.solve(ode, Tsit5(), abstol=abstol, reltol=reltol, saveat=saveat)

    t = sol.t
    x = t -> sol(t)[1]
    y = t -> sol(t)[2]
    θ = t -> sol(t)[3]

    return t, x, y, θ

end;
```

In [6]:
```
# trajectoire réaliste
t, x, y, θ = realistic_trajectory(tf, t0, x0, y0, θ0, u, current)

# ------------------------------------------------------------------------------------------------------------
# affichage

# trajectoire
plt_q = plot(xlims=(-2, 6), ylims=(-1, 8), aspect_ratio=1, xlabel="x", ylabel="y")
plot_state!(plt_q, x0, y0, θ0; color=2)
plot_state!(plt_q, xf, yf, θf; color=2)
plot_current!(plt_q; current=current)
plot_trajectory!(plt_q, t, x, y, θ)
plot_state!(plt_q, x(tf), y(tf), θ(tf); color=3)

# contrôle
plt_u = plot(t, u; color=1, legend=false, linewidth=2, xlabel="t", ylabel="u")

#
plot(plt_q, plt_u;
    layout=(1, 2),
    size=(1200, 600),
    leftmargin=5mm,
    bottommargin=5mm,
    plot_title="Simulation avec modèle de courant"
)
```

Out[6]:



Approche MPC

En pratique, nous n'avons pas à l'avance les données réelles du courant sur l'ensemble du trajet, c'est pourquoi nous allons recalculer régulièrement le contrôle optimal. L'idée est de mettre à jour le contrôle optimal à intervalle de temps régulier en prenant en compte le courant à la position où le navire se trouve. On est donc amener à résoudre un certain nombre de problème à courant constant, avec celui-ci mis réguilièremet à jour. Ceci est une introduction aux méthodes dites MPC, pour "Model Predictive Control" en anglais.

In [7]:
```
Nmax = 20   # nombre d'itérations max de la méthode MPC
ε = 1e-1    # rayon sur la condition finale pour stopper les calculs
Δt = 1.0    # pas de temps fixe de la méthode MPC
P = 300      # nombre de points de discrétisation pour le solveur

t1 = t0
x1 = x0
y1 = y0
θ1 = θ0

data = []

N = 1
stop = false

while !stop

    # on récupère le courant à la position actuelle
    w = current(x1, y1)

    # on résout le problème
    t, x, y, θ, u, tf, iter, cons = solve(t1, x1, y1, θ1, xf, yf, θf, w; grid_size=P, display=false);

    # calcul du temps suivant
    if (t1+Δt < tf)
        t2 = t1+Δt
    else
        t2 = tf
        println("t2=tf: ", t2)
        stop = true
    end

    # on stocke les données: le temps initial courant, le temps suivant, le contrôle
    push!(data, (t2, t1, x(t1), y(t1), θ(t1), u, tf))
```

```
    # on met à jour les paramètres de la méthode MPC: on simule la réalité
    t, x, y, θ = realistic_trajectory(t2, t1, x1, y1, θ1, u, current)
    t1 = t2
    x1 = x(t1)
    y1 = y(t1)
    θ1 = θ(t1)

    # on calcule la distance à la position cible
    distance = norm([ x1, y1, θ1 ] - [ xf, yf, θf ])
    println("N: ", N, "\t distance: ", distance, "\t iterations: ", iter, "\t constraints: ", cons, "\t tf: ",
    if !((distance > ε) && (N < Nmax))
        stop = true
    end

    #
    N += 1

end
```

```
N: 1     distance: 7.169372990138332     iterations: 42  constraints: 9.122780308956635e-10      tf: 9.96925102
6991449
N: 2     distance: 6.619012387410927     iterations: 45  constraints: 2.5287216764979803e-10     tf: 11.5842488
85909826
N: 3     distance: 6.811824990282241     iterations: 25  constraints: 5.462011398726929e-11      tf: 12.1924074
7530206
N: 4     distance: 6.805743942554834     iterations: 63  constraints: 2.4345414573190283e-10     tf: 12.4935984
38558452
N: 5     distance: 6.836922010269605     iterations: 47  constraints: 4.88927343056389e-10       tf: 12.6759957
1922167
N: 6     distance: 6.904144875436707     iterations: 25  constraints: 7.912370758589304e-11      tf: 12.8107056
58643144
N: 7     distance: 7.0070773287165915    iterations: 35  constraints: 9.789671850946036e-10      tf: 12.9110876
54424756
N: 8     distance: 7.14047407623855      iterations: 40  constraints: 1.5706123901448876e-11     tf: 12.9903416
04334535
N: 9     distance: 6.682527995280143     iterations: 44  constraints: 7.531752999057062e-13      tf: 13.3844841
74111117
N: 10    distance: 5.585966710664459     iterations: 36  constraints: 0.015091307541053478       tf: 13.3740842
45407065
N: 11    distance: 3.9736943069307467    iterations: 60  constraints: 0.012697863545896193       tf: 13.3545700
39130135
N: 12    distance: 2.1132291368980236    iterations: 20  constraints: 2.224759043656377e-9       tf: 13.3117697
96606038
N: 13    distance: 0.41968650669472385   iterations: 41  constraints: 2.3105551177238226e-5      tf: 13.2948542
07525157
t2=tf: 13.305053260732876
N: 14    distance: 0.03134653710002621   iterations: 85  constraints: 0.0001017474278581787      tf: 13.3050532
60732876
```

Affichage

In [8]:

```
# trajectoire
plt_q = plot(xlims=(-2, 6), ylims=(-1, 8), aspect_ratio=1, xlabel="x", ylabel="y")

# condition finale
plot_state!(plt_q, xf, yf, θf; color=2)

# courant
plot_current!(plt_q; current=current)

# contrôle
plt_u = plot(xlabel="t", ylabel="u")

N = 1

for d ∈ data

    #
    t2, t1, x1, y1, θ1, u, tf = d

    # calcule de la trajectoire réelle
    t, x, y, θ = realistic_trajectory(t2, t1, x1, y1, θ1, u, current)

    # trajectoire
    plot_state!(plt_q, x1, y1, θ1; color=2)
    plot_trajectory!(plt_q, t, x, y, θ; N=0)

    # contrôle
    plot!(plt_u, t, u; color=1, legend=false, linewidth=2)
```

```
    N += 1

end

plot_state!(plt_q, x(tf), y(tf), θ(tf); color=3)

#
plot(plt_q, plt_u;
    layout=(1, 2),
    size=(1200, 600),
    leftmargin=5mm,
    bottommargin=5mm,
    plot_title="Simulation avec modèle de courant"
)
```

Out[8]: