TÉCNICO LISBOA

Route Distance & Dead-Reckoning Navigation

1º Trabalho Laboratorial

Carolina Serra 80768
Tiago Alexandre 81024
João Gonçalves 81040

Instituto Superior Técnico
Mestrado em Engenharia Aeroespacial
Sistemas Aviónicos Integrados

Outubro de 2018

Conteúdo

Resumo

Neste trabalho é desenvolvido, otimizado, testado e avaliado um algoritmo para o cálculo da distância entre dois *waypoints* na Terra, e desenvolvido e analisado um simulador de navegação *Dead Reckoning*.

1 Introdução

O Flight Management System é um sistema aviónico que permite reduzir significativamente a carga de trabalho do piloto, uma vez que, através da integração e do processamento dos sinais GPS, dos sinais das radio-ajudas e dos sinais provenientes sensores de navegação inercia, determina a posição da aeronave e guia a mesma segundo a rota definida no plano do voo.

Este trabalho está dividido em duas partes, ambas implementações de funcionalidades deste sistema. A primeira parte foca-se em calcular a distância total percorrida num conjunto de *waypoints*, recorrendo ao conceito de distância ortodrómica. A segunda parte simula um sistema de navegação *Dead Reckoning*, em que a partir de apenas um sensor de velocidade-ar, a posição ao longo do tempo é atualizada e as referências de navegação são calculadas.

Definem-se também as simplificações empregues, os conceitos científicos, e descreve-se brevemente a arquitetura do software desenvolvido. Finalmente, expõe-se os procedimentos de teste que permitem validar esta solução, e analisam-se resultados.

Em anexo está presente toda a documentação do código C produzido.

2 Simplificações Empregues

O algoritmo implementado tem em consideração alguns aspetos importantes:

1. A superfície da Terra é considerada esférica e pretende-se calcular a distância absoluta entre dois *waypoints*, que corresponde à distância ortodrómica entre ambos os pontos.
2. Os *waypoints* são definidos em termos de latitude, longitude e altitude, e os seus valores são expressos em conformidade com o formato aeronáutico standard.
3. A altitude varia ao longo da rota, mas entre dois *waypoints* consecutivos é considerada constante e definida pela altitude do primeiro *waypoint*.

Os fundamentos teóricos inerentes ao cálculo da distância ortodrómica são apresentados na secção seguinte. As simplificações realizadas nas alíneas 1 e 3 têm um erro associado que será discutido e numericamente estimado na secção de teste e avaliação do algoritmo.

Para a segunda parte deste trabalho, acrescenta-se:

1. Não se considera a existência de vento, ie $TAS \equiv GS$.
2. A superfície da Terra pode ser considerada plana numa vizinhança da ordem de 10 km.

3 Princípios científicos

3.1 Distância ortodrómica

A distância ortodrómica corresponde ao comprimento da trajetória mais curta entre dois pontos de uma superfície esférica e é aquela que, de forma aproximada, é percorrida pelas aeronaves quando se deslocam entre dois *waypoints* consecutivos. Para dois pontos não antipodais numa superfície esférica, esta distância corresponde ao comprimento do segmento mais curto do círculo máximo que os une, que pode ser obtido com base na definição de produto interno.

Atendendo a que $(\phi_A\,,\,\lambda_A)$ e $(\phi_B,\,\lambda_B)$ correspondem ao par (latitude, longitude) dos pontos A e B respetivamente, que $R = ||\overrightarrow{r_A}|| = ||\overrightarrow{r_B}||$ corresponde à distância destes pontos ao centro da Terra e tendo em conta

o esquema vetorial da Figura X e o sistema de coordenadas cartesiano geodésico $(\overrightarrow{e_x}, \overrightarrow{e_y}, \overrightarrow{e_z})$ representado, temos que:

$$\overrightarrow{r_A} = R \begin{pmatrix} \cos\phi_A \cos\lambda_A & \cos\phi_A \sin\lambda_A & \sin\phi_A \end{pmatrix} \begin{pmatrix} \overrightarrow{e}_x & \overrightarrow{e}_y & \overrightarrow{e}_z \end{pmatrix}^T$$
$$\overrightarrow{r_B} = R \begin{pmatrix} \cos\phi_B \cos\lambda_B & \cos\phi_B \sin\lambda_B & \sin\phi_B \end{pmatrix} \begin{pmatrix} \overrightarrow{e}_x & \overrightarrow{e}_y & \overrightarrow{e}_z \end{pmatrix}^T$$

O ângulo α entre estes dois vetores pode ser obtido pelo produto interno de ambos:

$$\overrightarrow{r_A} \cdot \overrightarrow{r_B} = ||\overrightarrow{r_A}|| \cdot ||\overrightarrow{r_B}|| \cos\alpha$$
$$\Rightarrow \alpha = \cos^{-1} \frac{\vec{r_A}.\vec{r_B}}{||\overrightarrow{r_A}|| \cdot ||\overrightarrow{r_B}||}$$

$$= \cos^{-1} \left(\frac{R^2(\cos\phi_A \cos\lambda_A \cos\phi_B \cos\lambda_B + \cos\phi_A \sin\lambda_A \cos\phi_B \sin\lambda_B + \sin\phi_A \sin\phi_B)}{R^2} \right)$$
$$= \cos^{-1} (\sin\phi_A \sin\phi_B + \cos\phi_A \cos\phi_B \cos(\lambda_B - \lambda_A))$$

Pelo que o arco de circunferência entre os dois pontos - que corresponde à distância ortodrómica d – é facilmente obtido pela seguinte expressão, com α em radianos:

$$d = R\alpha$$

O rumo inicial é dado por:

$$\chi = \arccos \left(\frac{\cos\phi_A \sin\phi_B - \cos\phi_B \sin\phi_A \cos(\lambda_B - \lambda_A)}{\sin\alpha} \right) \tag{1}$$

À medida que a aeronave se desloca, o ponto A pode ser actualizado e usado novamente na mesma expressão para actualizar chi.

3.2 Integração da velocidade

O sistema de navegação *Dead-reckoning* integra a velocidade da aeronave, com base na TAS e nas referências dos ângulos de rumo χ_{ref} e subida θ_{ref}, medidos por sensores a bordo. O programa usa para o passo de integração um Δt que corresponde a uma distância percorrida de 10 km, tal que a curvatura da Terra possa ser negligenciada. A distância percorrida no plano horizontal (s) é:

$$V_N = TAS \cdot \cos\theta_{ref} \cdot \cos\chi_{ref}$$
$$V_E = TAS \cdot \cos\theta_{ref} \cdot \sin\chi_{ref}$$
$$s = \sqrt{V_N^2 + V_E^2} \cdot \Delta t.$$

Dado que a velocidade vertical é modelada por:

$$\dot{h} = \rho \left(h_{ref} - h \right),$$

O ângulo θ_{ref} necessário pode ser calculado por:

$$\theta_{ref} = \arcsin \left(\frac{\rho(h_{ref} - h)}{TAS} \right), \tag{2}$$

enquanto que χ_{ref} é calculado pela equação 1.

O círculo maior é dividido em troços, que são percorridos com rumo constante, ou seja, em rotas loxodrómicas. O ponto B' que é alcançado quando é percorrida a distância s com rumo χ_{ref} segue as equações [3]:

$$\begin{aligned} \delta &= \frac{s}{R+h} \\ \phi_{B'} &= \phi_A + \delta \cos \chi_{ref} \\ \lambda_{B'} &= \lambda_A + \tan \chi_{ref} \cdot \ln \left(\frac{\tan\left(\pi/4 + \phi_{B'}/2\right)}{\tan\left(\pi/4 + \phi_A/2\right)} \right) \end{aligned} \tag{3}$$

3.3 Controlador de velocidade

Para alguns problemas, a velocidade da aeronave é controlada em função de uma medida de velocidade com ruído, segundo o esquema genérico da figura 1.

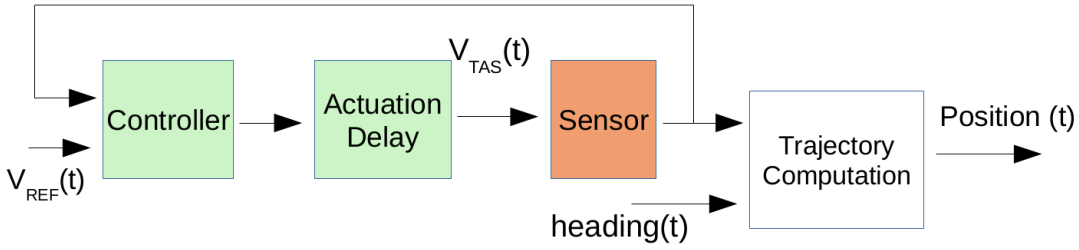

Figura 1: Esquema de controlo de velocidade.

A implementação escolhida admite que, por conveniência, o controlador use o mesmo período de amostragem que o resto do sistema de navegação, que será na ordem dos 50 s, embora se reconheça que este seja demasiado longo considerando o período do ruído. Implementa-se um controlo integral. Em tempo discreto, tal é descrito pela equação 4. Introduzindo um atraso unitário de atuação, ficamos com a equação 5.

$$TAS[k] = TAS[k-1] + \left(V_{ref} - TAS_m\right)[k] \tag{4}$$

$$TAS[k] = TAS[k-1] + \left(V_{ref} - TAS_m\right)[k-1] \tag{5}$$

Adiciona-se o ruído do sensor, e o subsistema passa a usar o esquema de controlo da figura 2.

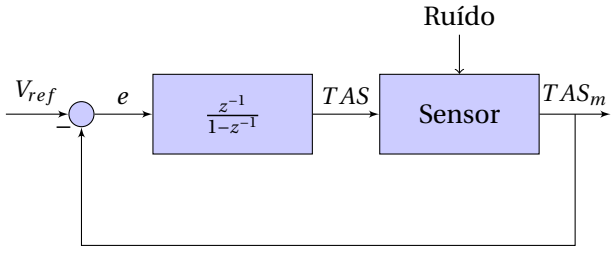

Figura 2: Controlador implementado.

3.4 Representação de coordenadas geográficas

A representação de coordenadas geográficas nos ficheiros que descrevem as rotas seguem a convenção **ISO_6709**. As características são:

- Separação de latitude, longitude e altitude com espaços,

- Formato graus, minutos e segundos decimais,
- **N**, **S**, **E**, **O** imediatamente a seguir aos dígitos a que dizem respeito,
- altitude especificada com unidades (usados `ft` no programa),
- separador decimal configurado pelo sistema operativo (usado o ponto <.> no programa).

Exemplos:

```
50°40'46,461"N 95°48'26,533"W 123.45m
20°22'41.880000"S 118°37'22.800000"E 35000ft
```

No programa, incluem-se a o *id* do *waypoint*, e na segunda parte, a velocidade TAS, separados por <;>, como por exemplo:

```
PUGUT;27°43'2.280000"S 131°53'13.200000"E 35000ft;
11-LONDON;51°30'35.5140"N 0°07'5.1312"W 30000ft;700km/h;
```

4 Eficiência Numérica

A fórmula deduzida para o cálculo do ângulo α entre os dois vetores $\vec{r}_1$ e $\vec{r}_2$ que unem cada um dos pontos A e B ao centro do sistema de coordenadas, não é bem condicionada quando a distância entre os dois pontos é pequena - nestes casos a computação do *arccos* tem um erro de arredondamento significativo. Apesar de na aviação serem raros os casos em que dois *waypoints* distam apenas uns quilómetros entre si e apesar de na implementação deste algoritmo se utilizar precisão dupla como resposta a tais erros de arredondamento, optou-se mesmo assim por utilizar a fórmula para o cálculo do ângulo α a fórmula de *V*icenty [1], que é bem condicionada para quaisquer distâncias entre dois *waypoints*:

$$\alpha = \arctan \frac{\sqrt{(\cos\phi_B \sin(\lambda_B - \lambda_A))^2 + (\cos\phi_A \sin\phi_B - \sin\phi_A \cos\phi_B \cos(\lambda_B - \lambda_A))^2}}{\sin\phi_A \sin\phi_B + \cos\phi_A \cos\phi_B \cos(\lambda_B - \lambda_A)}$$

A distância ortodrómica passa a ser obtida pela seguinte expressão final:

$$d = \left(\arctan \frac{\sqrt{(\cos\phi_B \sin(\lambda_B - \lambda_A))^2 + (\cos\phi_A \sin\phi_B - \sin\phi_A \cos\phi_B \cos(\lambda_B - \lambda_A))^2}}{\sin\phi_A \sin\phi_B + \cos\phi_A \cos\phi_B \cos(\lambda_B - \lambda_A)} \right) R, \tag{6}$$

que é a expressão usada no programa, na função `route_distance`.

A equação 3 que calcula a nova longitude quando a aeronave se desloca com rumo fixo é mal condicionada quando o rumo é ±90°, caso em que a aeronave se move diretamente para Este ou Oeste. Neste caso temos uma indeterminação do tipo $0 \times \infty$. Para evitar isso, segue-se o seguinte algoritmo [3]:

$$\begin{aligned} \Delta\psi &= \ln\left(\frac{\tan(\pi/4 + \phi_{B'}/2)}{\tan(\pi/4 + \phi_A/2)}\right) \\ q &= \frac{\phi_{B'} - \phi_A}{\Delta\psi} \quad \text{se} \quad \Delta\psi > 10 \times 10^{-9}, \quad \text{c.c.} \quad \cos\phi_A \\ \lambda_{B'} &= \lambda_A + \frac{\delta \cdot \sin\chi_{ref}}{q}. \end{aligned} \tag{7}$$

5 *Software Design*

5.1 *Flight Management System*

As funcionalidades do *Flight Management System* implementadas são inseridas na biblioteca dinâmica `libfms.so`, compilada dos ficheiros `fms.h` e `fms.c`. A documentação detalhada destes ficheiros é incluída nos anexos C e B. Em síntese, as funções desenvolvidas são:

`route_distance`

Usa a equação 6 sobre um conjunto de *waypoints* e devolve a distância total percorrida.

`import`, `import_with_velocity`

Abre o ficheiro de *input* e carrega os *waypoints* para memória.

`next_position`

Usa o algoritmo 7 para integrar a velocidade da aeronave e calcular a nova posição.

`heading_angle`

Usa a equação 1 para calcular o ângulo de rumo (χ_{ref}) para qualquer instante do tempo.

`climb_angle`

Usa a equação 2 para calcular o ângulo de subida (θ_{ref}) para qualquer instante do tempo.

`log_data`, `log_errors`

Ferramentas para documentar as trajetórias e seus erros.

5.2 main

Com estas ferramentas, desenvolve-se um programa que resolve os problemas propostos, em `main.c` (anexo D).

A primeira parte do programa recebe o nome do ficheiro de *input* da linha de comandos, e calcula a distância total, imprimindo no terminal.

```
61          i = import(argv[1], points, 100);
62
63          result = route_distance(points, i);
```

```
67                      printf("Total route distance for file %s: %lfnm\n", argv[1],
68                                                              result);
```

O programa não usa qualquer forma de memória dinâmica, de modo a garantir segurança. No excerto acima, o vetor `points` faz parte do *Data Segment*, garantindo-se a segurança de acesso guardando o número máximo (fixo) de posições. No código pode ver-se que este valor é 100.

A segunda parte do programa importa a rota predefinida e simula o comportamento de um sistema de navegação *Dead-Reckoning*, escrevendo ficheiros com todas as variáveis ao longo do tempo e registando o erro entre os vários métodos. O excerto presente no anexo E foca-se no cálculo da trajetória simples que corresponde ao ponto 1 do enunciado.

Para cada troço, é lida a velocidade e as coordenadas do *waypoint*, e define-se o passo de integração conforme descrito na secção 3.2. De seguida procede-se à navegação até se chegar ao destino. São calculados os ângulos de referência (143-144), calcula-se a posição seguinte (145) e registam-se as variáveis (147). No final, é calculado o erro no *waypoint* (216-219).

As linhas (137-140) dizem respeito ao controlador da TAS descrito na secção 3.3.

5.3 *output*

A parte 1 do programa exibe a distância percorrida no terminal.

A parte 2 escreve ficheiros com a maioria dos dados calculados, são estes o tempo, posição, ângulo de rumo, subida e velocidade, assim como alguns erros. Eis a descrição dos vários ficheiros:

- `log_simple.txt` – Trajetória simples, definida pelo ponto 1 do enunciado.
- `log_noisy.txt` – Trajetória calculada com o sensor com ruído, definido pelo ponto 1 do enunciado.
- `log_err_noisy.txt` – Erro ao longo do tempo da trajetória anterior, comparada com a trajetória simples.
- `log_feedback.txt` – Trajetória calculada com o esquema de controlo da figura 1, ponto 3 do enunciado.
- `log_err_feedback.txt` – Erro da mesma comparada com a trajetória simples.
- `log_err_feedback-self.txt` – Erro da mesma quando comparada com a integração da velocidade **real**[1].
- `log_corrected.txt` – Trajetória calculada com o esquema da figura 1, e com correção da posição nos *waypoints*, ponto 4 do enunciado.
- `log_err_corrected.txt` – Erro da mesma em relação à trajetória simples.
- `log_err_corrected-self.txt` – Erro da mesma comparando com a integração da velocidade **real**.
- `log_err_waypoints.txt` – Erro de todos os métodos nos *waypoints*.

Exemplo do conteúdo de `log_corrected.txt`:

```
time;position;tas;heading_angle;climb_angle;
51.43s;38°48'37.060307"N 9°4'32.548194"W 30357.14ft;701.9km/h;35.45°;0.62°;
102.86s;38°53'0.920432"N 9°0'30.984758"W 30688.78ft;701.7km/h;35.49°;0.58°;
154.29s;38°57'24.539673"N 8°56'29.015940"W 30996.72ft;701.5km/h;35.53°;0.54°;
[...]
7479.41s;48°51'47.196910"N 2°20'47.979451"E 34999.90ft;700.1km/h;43.65°;0.00°;
Switched destination to waypoint 3 and corrected position
7530.84s;48°54'26.994272"N 2°27'38.642718"E 33916.46ft;699.9km/h;58.68°;-1.87°;
[...]
```

5.4 Compilação

A compilação está automatizada num `Makefile`. Para efectuar o *link* a outro programa qualquer, usa-se

```
gcc -o myprogram myprogram.o -L<path> -lfms -Wl,-rpath=<path>
```

onde `<path>` é a localização do ficheiro `libfms.so`.

5.5 Extra

Foi também desenvolvido um *script* em python para descarregar do website flightplandatabase.com uma rota identificada pelo seu URL, e escrevê-lo com o formato usado pelo programa. Este ficheiro está disponível em `web_parser/fetcher.py`.

[1] Note-se que a velocidade da aeronave varia com o tempo para este caso, portanto para o cálculo de erros deve usar-se o valor real da posição, que é obtido integrando a velocidade real ao longo do tempo, não simplesmente comparando com a trajetória simples.

6 Testes

Foram desenvolvidos testes unitários às funções de cálculo, usando o *framework* de testes C++ catch2, que é uma biblioteca *single header*, fácil de incluir no projeto.

Para a primeira parte, a distância da rota é comparada com a dada pelo site de onde a rota foi extraída, flightplandatabase.com. O excerto presente no anexo F exemplifica os testes criados para esta função. Alterar o valor da tolerância de erro para 0 implica que estes testes falhem, de onde se podem ver os erros percentuais:

```
  Route distance tests, vicenty's formula
  New Orleans - Paris
-------------------------------------------------------------------------------
tests-main.cpp:24
...............................................................................

tests-main.cpp:28: FAILED:
  CHECK( fabs(route_distance(points, c)-result)/result < error_tolerance )
with expansion:
  0.0015389467 < 0.0
```

Deste *output*, verifica-se que o erro percentual neste trajeto é 0.15 %.

Para a integração da velocidade que atualiza a posição, e para o cálculo do ângulo de rumo, os resultados são comparados com os dados pela ferramenta presente em [3].

Todos os testes estão presentes em `test/tests-main.cpp`.

```
===============================================================================
All tests passed (22 assertions in 5 test cases)
```

De maneira a verificar o correto uso de recursos, efectuaram-se testes de memória com uso do `Valgrind`. O resultado é:

```
==7239== HEAP SUMMARY:
==7239==     in use at exit: 0 bytes in 0 blocks
==7239==   total heap usage: 25 allocs, 25 frees, 59,872 bytes allocated
==7239==
==7239== All heap blocks were freed -- no leaks are possible
```

Como se verifica, não há perdas de memória ou acessos incorretos. Os *allocs* referidos provêm do uso de ficheiros.

Os testes de integração consistem na avaliação dos resultados obtidos, na secção seguinte.

7 Análise de resultados

Com o intuito de despistar erros grosseiros e garantir que as trajetórias coincidem com as da rota proposta no enunciado utilizou-se a biblioteca gratuita `m_map`[2] para Matlab, obtendo-se o seguinte percurso numa representação cilíndrica equidistante da Terra:

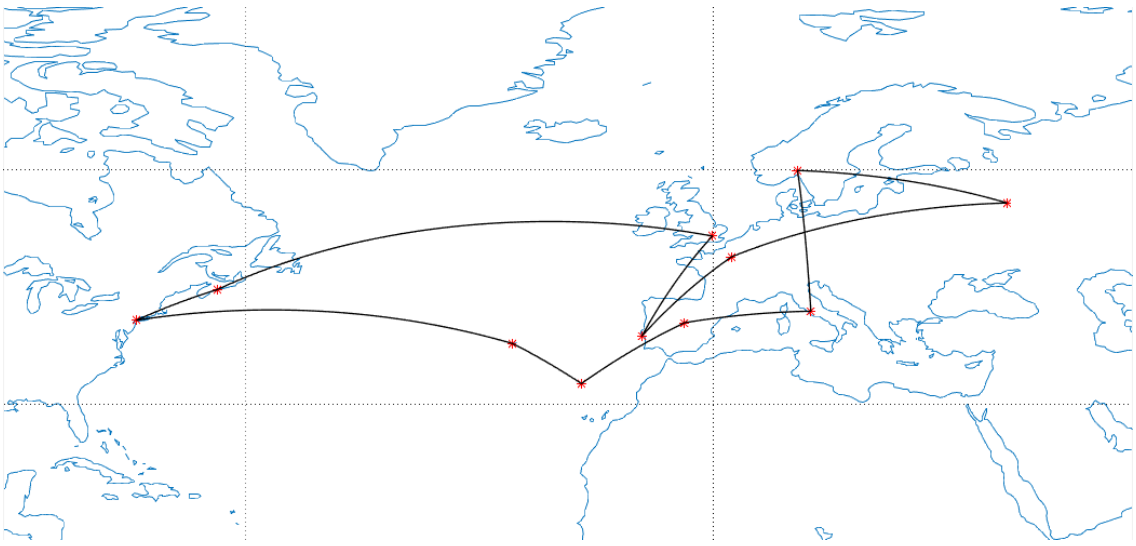

Figura 3: Rota descrita no ponto 1, importada de `log_simple.txt`

Pela figura confirma-se que o conjunto de curvas geradas são semelhantes a grandes círculos e as ligações representadas correspondem às da rota que nos foi proposta.

De forma a poder fazer uma análise e uma comparação cuidada das trajetórias determinadas neste trabalho laboratorial, representou-se no gráfico da figura 4 a evolução do erro entre a rota simples e:

- A rota calculada considerando ruído na medição da velocidade (correspondente à linha laranja na figura).
- A rota calculada com um auto piloto simples implementado com o objetivo de compensar o erro do sensor de velocidade (correspondente à linha azul).
- A rota que para além do auto piloto ainda possuí uma correção no momento de chegada a cada um dos *waypoints* (correspondente à linha preta).

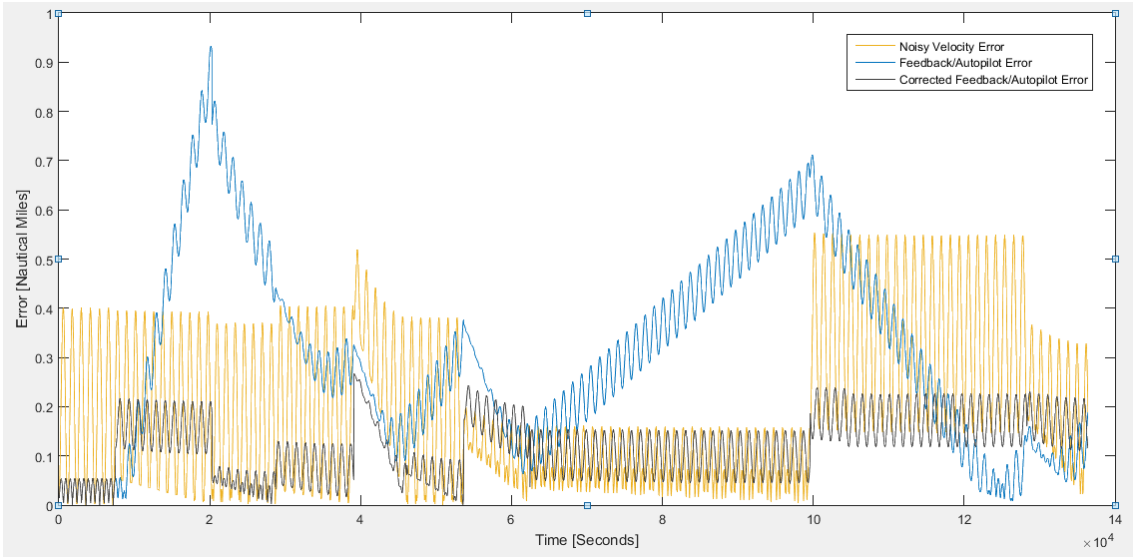

Figura 4: Diferença entre a posição da trajetória simples sem erros de sensor e cada uma das três rotas acima descritas.

O primeiro aspeto que se consegue inferir é que, como era expectável, o ruído do sensor de velocidade tem a forma de um seno e não acumula ao longo do tempo, o que justifica o facto de não termos curvas estritamente crescentes. Quando se introduz erros do sensor na medição da velocidade (curva a laranja) obtêm-se diferenças com a posição da trajetória simples nunca superiores a 0.6NM e com uma média ao longo de todo o percurso de 0.214NM. O auto piloto simples introduzido (curva a azul) apresentou uma oscilação significativa, introduzido em alguns intervalos um erro superior ao registado sem qualquer método de compensação de incertezas. O auto piloto apenas apresentou resultados realmente satisfatórios com a introdução da funcionalidade de correção da posição no momento de chegada a um *waypoint* (curva a preto), permitindo reduzir para sensivelmente metade o erro introduzido pelo sensor de velocidade, obtendo-se diferenças com a posição da trajetória simples nunca superiores a 0.3NM e com uma média ao longo de todo o percurso de 0.123NM.

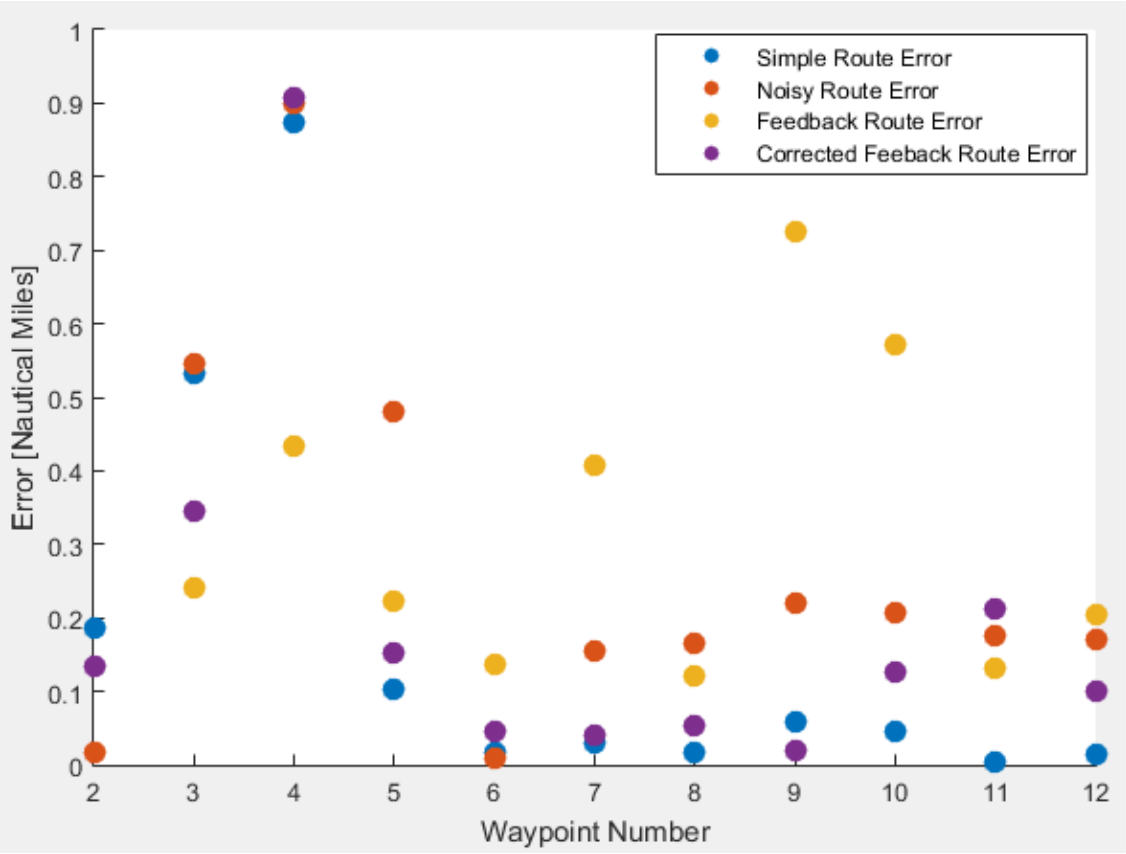

Figura 5: Erro nos *waypoints* dos vários métodos de cálculo.

Na Figura 5 é possível aferir o erro de cada uma das trajetórias (simples, com erro do sensor de velocidade, com autopiloto simples e com autopiloto com correção na chegada a um *waypoint*) com as coordenadas reais de cada *waypoint*.

Para além das conclusões acima retiradas, como a que constata que a correção da posição no momento de chegada a um waypoint diminuí significativamente o erro do auto piloto e a de que este permite compensar parte do erro introduzido pelo sensor de velocidade, também é possível concluir da análise deste gráfico que a trajetória simples não coincide exatamente com as coordenadas reais de cada um dos *waypoints* porque tem associada erros derivados da integração da velocidade. No entanto, e como era de esperar, esta é tendencialmente a trajetória com menor erro nas medições nos *waypoints*.

8 Conclusão

No final deste trabalho concluímos que um sensor com estas características é suficiente para navegar grandes distâncias, mas reconhecemos que numa situação real o vento é uma variável crucial a ter em conta para que o sistema funcione corretamente.

Mesmo sem um sistema de localização externo continuamente disponível, é possível compensar os erros que se acumulam corrigindo a posição em pontos de referência, como nas proximidades de um VOR/DME. Desta maneira, o último caso (o autopiloto com correção da posição nos *waypoints*) é o que apresenta melhor desempenho, tendo o menor erro quando considerado o ruído do sensor.

Referências

[1] Thaddeus Vincenty. «Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations». Em: *Survey review* 23.176 (1975), pp. 88–93.

[2] E. Firing et al. *M_Map: A Mapping package for Matlab*. URL: `https://www.eoas.ubc.ca/~rich/map.html` (acedido em 31/10/2018).

[3] *Calculate distance, bearing and more between Latitude/Longitude points*. URL: `https://www.movable-type.co.uk/scripts/latlong.html?from=48.9613600,-122.0413400&to=48.965496,-122.072989` (acedido em 15/10/2018).

Anexos

A waypoints Struct Reference

Greographical position.
`#include <fms.h>`

Data Fields

- double height
- double latitude
- double longitude

A.1 Detailed Description

Greographical position.

A.2 Field Documentation

height

`double waypoints::height`
Height, in feet.

latitude

`double waypoints::latitude`
Latitude, in radians.

longitude

`double waypoints::longitude`
Longitude, in radians.

B L1/fms/fms.c File Reference

Implementation file for the Flight Management System Library.

Include dependency graph for fms.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "fms.h"
```

C L1/fms/fms.h File Reference

Header file for the Flight Management System library.

Data Structures

- struct waypoints
  Greographical position.

Macros

- #define **ALPHA** 5

Functions

- double route_distance (struct waypoints *points, int count)
  Route distance of a set of waypoints.
- double route_distance_v0 (struct waypoints *points, int count)
  Route distance of a set of waypoints, version 0.
- int import (char *file, struct waypoints *points, size_t max_size)
  import route from file.
- int import_with_velocity (char *file, struct waypoints *points, double *velocities, size_t max_size)
  import a route from file, with velocity column
- struct waypoints next_position (struct waypoints previous, double tas, double psi_ref, double theta_ref, double delta_t)
  Compute next position given a set of references.
- void wrap_pi (double *angle)
  wraps an angle to the interval [-pi +pi]
- double heading_angle (struct waypoints pos1, struct waypoints pos2)
  computes reference true heading angle for great-circle navigation
- double climb_angle (double h1, double h_ref, double tas)
  computes reference climb angle.
- void log_data (FILE *os, struct waypoints pos, double t, double heading, double climb, double tas)
  Logs one line position data to file.
- void log_errors (FILE *os, double error, double t)
  Logs one line of error data to file.

C.1 Detailed Description

Header file for the Flight Management System library.

Author

João Gonçalves, Tiago Oliveira, Carolina Serra

Date

31 Oct 2018

Defines functions for computing route distances, navigation references, and contains I/O utilities. Can be linked to C or C++ programs.

See also

`https://github.com/jomigo96/SAI2018`

C.2 Function Documentation

C.2.1 climb_angle()

```
double climb_angle (
          double h1,
          double h_ref,
          double tas )
```

computes reference climb angle.
Follows the desired dynamics: $\dot{h} = \alpha(h_{ref} - h)$
Does not consider wind.

Parameters

h1	current height, in feet.
h_ref	reference height, in feet.
tas	True Air Speed, in km/h.

Returns

Climb angle, in radian.

C.2.2 heading_angle()

```
double heading_angle (
          struct waypoints pos1,
          struct waypoints pos2 )
```

computes reference true heading angle for great-circle navigation
This is also called initial bearing, the heading the aircraft must have at `pos1` to follow a great-circle route to `pos2`.

Parameters

pos1	Current position
pos2	Destination

Returns

Reference heading, in radian.

C.2.3 import()

```
int import (
          char * file,
          struct waypoints * points,
          size_t max_size )
```

import route from file.

Imports a route from a file, with example format:
KADOM;33°42'38.880000"S 150°18'0.000000"E 35000ft;

Parameters

file	path to file.
points	pointer to the array which will hold the data.
max_size	maximum size of the array, after which points are not imported.

Returns

number of points imported, -1 in case of error.

C.2.4 import_with_velocity()

```
int import_with_velocity (
          char * file,
          struct waypoints * points,
          double * velocities,
          size_t max_size )
```

import a route from file, with velocity column
Identical to `import()`, but the last column contains velocity (TAS):
7-FUNCHAL;32°37'59.99"N 16°53'60.00"W 30000ft;400km/h;

Parameters

file	path to file.
points	pointer to the array which will hold the position data.
velocities	pointer to the array which will hold velocities.
max_size	maximum size of both arrays, after which points are not imported.

Returns

number of points imported, -1 in case of error.

C.2.5 log_data()

```
void log_data (
          FILE * os,
          struct waypoints pos,
          double t,
          double heading,
          double climb,
          double tas )
```

Logs one line position data to file.
Data is written as this example:
977.14s;40°7'12.256353"N 7°50'58.295191"W 33776.90ft;700.0km/h;36.23°;0.16°;

Parameters

os	file pointer to open file.
pos	Position.
t	time in seconds.
heading	heading angle in radian.
climb	Climb angle in radian.
tas	True Air Speed in km/h.

C.2.6 log_errors()

```
void log_errors (
          FILE * os,
          double error,
          double t )
```

Logs one line of error data to file.

Data is written as this example: 668.57s;0.3765nm;

Parameters

os	file pointer to open file.
error	Position error, in nautical miles.
t	time in seconds.

C.2.7 next_position()

```
struct waypoints next_position (
          struct waypoints previous,
          double tas,
          double psi_ref,
          double theta_ref,
          double delta_t )
```

Compute next position given a set of references.

Calculates position at the next time-step, assuming all references are held constant in this time (zero-order hold). The sampling time must be small (up to 20km of distance travelled), since this function considers a locally flat Earth. Does not consider wind. The calculation is made with the formulas for rhumb lines (constant heading).

Parameters

previous	position at the current time-step
tas	current True Air Speed in km/h
psi_ref	True heading angle in radians
theta_ref	Reference climb angle in radians
delta_t	Sampling time in seconds

Returns

Position at the next time-step

C.2.8 route_distance()

```
double route_distance (
          struct waypoints * points,
          int count )
```

Route distance of a set of waypoints.

Computes the route distance of a set of waypoints, navigated following great-circle arcs (the smallest distance between two points in the surface of the Earth). Uses the spherical Earth, approximation, and Vicenty's method.

Parameters

points	pointer to an ordered array of waypoints.
count	number of waypoints in `points`.

Returns

Total route distance, in nautical miles.

C.2.9 route_distance_v0()

```
double route_distance_v0 (
          struct waypoints * points,
          int count )
```

Route distance of a set of waypoints, version 0.

Computes the route distance of a set of waypoints, navigated following great-circle arcs (the smallest distance between two points in the surface of the Earth). Uses the spherical Earth, approximation, and the standard acos formula. This function is deprecated, prefer `route_distance()`

Parameters

points	pointer to an ordered array of waypoints.
count	number of waypoints in `points`.

Returns

Total route distance, in nautical miles.

C.2.10 wrap_pi()

```
void wrap_pi (
          double * angle )
```

wraps an angle to the interval [-pi +pi]

Parameters

angle	pointer to the angle to be wrapped, also in radians.

D L1/fms/main.c File Reference

Program to solve laboratory work questions.

Include dependency graph for main.c:

```
#include <stdio.h>
#include <math.h>
#include "fms.h"
```

Functions

- int **main** (int argc, char **argv)

D.1 Detailed Description

Program to solve laboratory work questions.

Author

João Gonçalves, Tiago Oliveira, Carolina Serra

Date

31 Oct 2018

Part 1 computes the total distance of a route defined in a .txt file. The file is passed as a command line argument.

Part 2 assembles a Dead-reckoning navigation system.

See also

https://github.com/jomigo96/SAI2018

E Excerto de `main.c`

```
81          i = import_with_velocity(file, points, velocity, 100);

116         for(j=1; j<i; j++){
117
118                 tas = velocity[j-1];
119                 dt = 10.0/tas*3600.0;
120                 pos_pair[0]=points[j-1];
121                 pos_pair[1]=points[j];
122                 result = route_distance(pos_pair, 2) * nm2km;
123                 interval = ceil(result/(tas*dt/3600.0));
124                 last_dt = 3600.0*(result-(interval-1)*tas*dt/3600.0)/tas;

128                 while(interval>0){
```

```
                        t += dt;
                        tas_noisy = tas*(1+0.01*sin(2*M_PI*t/(20*60)));

                        tas_feedback = (tas_feedback + velocity[j-1] - vm_feedback);
                        // uses previous sample of vm_feedback

                        vm_feedback = tas_feedback*(1+0.01*sin(2*M_PI*t/(20*60)));

                        // Exact route (Point 1)
                        heading = heading_angle(current_pos, points[j]);
                        climb = climb_angle(current_pos.height, points[j].height, tas);
                        current_pos = next_position(current_pos, tas, heading, climb,
                                                    dt);
                        log_data(out1, current_pos, t, heading, climb, tas);

                        interval--;
                }

                pos_pair[0]=points[j];
                pos_pair[1]=current_pos;
                result = fabs(route_distance(pos_pair, 2));
                fprintf(error_waypoints, "Simple route"
                                "                                    : %.6lfnm\n", result);

        }
```

F Testes ao cálculo da distância

```
TEST_CASE("Route distance tests, vicenty's formula"){


        char file[40];
        double result;
        struct waypoints points[150];
        int c;

        SECTION("Unexisting file"){
                strcpy(file, "nofile.txt");

                // Should return -1 when file doesn't exist
                CHECK( import(file, points, 150) == -1 );
        }
        SECTION("New Orleans - Paris"){
                strcpy(file, "../routes/kmsy-lfpg.txt");
                result=4227.0;
                c=import(file, points, 150);
                CHECK(fabs(route_distance(points, c)-result)/result < error_tolerance);
        }


        SECTION("Chicago - Lisbon"){
                strcpy(file, "../routes/kord-lppt.txt");
                result=3748.0;
                c=import(file, points, 150);
```

```
36                  CHECK(fabs(route_distance(points, c)-result)/result < error_tolerance);
37          }
38
39          SECTION("Auckland-Doha"){
40                  strcpy(file, "../routes/nzaa-otbd.txt");
41                  result=7871.0;
42                  c=import(file, points, 150);
43                  CHECK(fabs(route_distance(points, c)-result)/result < error_tolerance);
44          }
45
46
47          SECTION("Shanghai-Los Angels"){
48                  strcpy(file, "../routes/zspd-klax.txt");
49                  result=5778.0;
50                  c=import(file, points, 150);
51                  CHECK(fabs(route_distance(points, c)-result)/result < error_tolerance);
52          }
53  }
```