Projeto 2 de Física Computacional

Pedro Bicudo e Nuno Cardoso , IST,

27 de Outubro de 2017

Resumo das instruções

Este projeto deve ser submetido como um único ficheiro comprimido, na página pessoal de aluno no fénix:

- no menu superior devem selecionar estudante
- no menu lateral devem selecionar submeter, projetos;
- deve surgir o grupo desta disciplina de FC, e lá podem submeter projetos e visualizar os projetos submetidos.

Este ficheiro comprimido, por exemplo .zip, deve incluir os vários ficheiros criados,

- usando as extensões .cpp e .h para os códigos de g++,
- .nb para os códigos de `mathematica` (a enviar sem figuras),
- resultados na forma de gráficos em formato .pdf com labels,
- a memória descritiva em formato .txt (ver parágrafo seguinte).

para diminuir o número de ficheiros, não é necessário enviar os ficheiros de dados produzidos (ou de texto) .txt. Não devem ser incluídos os ficheiros executáveis.

É importante que enviem uma pequena memória descritiva, em texto simples, chamado readme_g#p#.txt, que inclua apenas

- o número e nome dos alunos do grupo
- o número do grupo
- qualquer comentário, observação ou análise pedidos no enunciado,
- a lista de ficheiros submetidos, e para cada ficheiro algumas palavras (no máximo meia dúzia) a descreve-lo, indicando condições iniciais quando necessário,
- e finalmente, apenas caso hajam vários ficheiros a linkar em conjunto (por exemplo várias funções e um header), a linha de comandos para os linkar, por exemplo algo do tipo g++ -o g09p2c1.o g09p2c1.cpp g09p2c2.cpp g09p2c3.cpp g09p1c1.h

Para sua própria organização os alunos/grupos devem designar os códigos que forem construindo por g#p2c#.cpp devendo substituir os # por números, sendo o primeiro # o número do seu grupo, o 2 refere-se ao projeto 2 e o segundo # o número do código (ex: g09p2c2.cpp será o segundo código que o grupo 09 realiza para este projeto. Deve usar uma designação semelhante para os outros ficheiros com outras extensões. O próprio ficheiro zip deve ser denominado g#p2.zip .

Serão avaliados,

- o valor dos resultados (os plots .pdf e os comentários no readme_g#p#.txt),
- a clareza dos resultados (de novo na forma de plots e comentários),
- a ausência de erros e de fugas de memória nos executáveis,
- se o código compila,
- a simplicidade e economia de recursos computacionais do código,
- a indentação e os comentários que ajudem ao entendimento e uso do código.

1 Resolver uma EDO de 2a ordem: lei de Newton uni-dimensional

1.a Oscilador

(4pt) Use como código inicial `g#p2c0.cpp` (onde `#` é o seu numero de grupo) o seu código já escrito e testado nas aulas que aplique o Runge-Kutta para um equação diferencial ordinária (EDO) de 2a ordem. Note que uma EDO de 2a ordem equivale a duas EDOs de 1a ordem acopladas.

Adapte então o seu código para um novo código `g#p2c1.cpp` que resolva a 2a Lei de Newton num caso particular: oscilador harmónico uni-dimensional amortecido e forçado, que estudou na disciplina de Mecânica e Ondas, sendo a força dada por,

$$F = -k\,x - \lambda\,v + F_0\,\sin(w_f\,t)$$

onde obviamente se aplica a 2a Lei de Newton.

Considere o caso particular dos parâmetros
m=1 Kg,
k= 1 N /m,
λ= .5 N s /m,
F_0 =.5 N.
Considere condições iniciais `x[0]` e `v[0]`à sua escolha (por exemplo de 1 m e de 1 m/s) bem como um número de passos `n_max` correspondente a alguns ciclos do sistema. Verifique que o passo `h` da iteração no tempo é suficientemente pequeno para os plots serem estáveis se mudarmos o passo para valores ainda menores. Aplique para três frequências angulares forçadas diferentes
$w_f = w_0/3$,
$w_f = w_0$,
$w_f = 3\,w_0$,
calculadas a partir da frequência do oscilador amortecido não forçado $w_0 = \sqrt{\frac{k}{m} - \left(\frac{\lambda}{2\,m}\right)^2}$.

Deve colocar todos os parâmetros e dados iniciais em ficheiros de leitura. São demasiados parâmetros para os inserir à mão do teclado sempre que vai testar o programa.

Num ficheiro de escrita, para depois analisar, coloque todos os dados relevantes em colunas: o índice de iteração `n` o tempo t a posição x, a velocidade `v`.

Com o `mathematica` desenhe 3 plots para cada caso: de $x(t)$, $v(t)$ e $v(x)$ (este dando o espaço das fases). Não se esqueça, como sempre em todos os projetos e alíneas, de apresentar o código para os plots (.nb para `mathematica` ou outro, mas sem gráficos), e (**importante**) o output dos gráficos em .pdf. Deve também ter labels com variáveis (e com as suas unidades físicas) nos eixos.

Observação: para retirar colunas de um array em `mathematica` afim de fazer os plots, pode usar o seguinte comando: se os dados estiverem num array chamado por exemplo `data`, com por exemplo 6 colunas, para ficar com apenas, por exemplo a 1a e a 3a colunas basta escrever `data[[All,{1,3}]]`.

Comente brevemente numa breve frase apenas da memória descritiva, comparando os resultados nos três casos particulares a frequência inicial e final com a respetiva frequência forçada w_f, e com a frequência do oscilador amortecido não forçado w_0.

1.b Genérico

(4pt) Pretendemos agora criar um código mais geral para resolver a 2a Lei de Newton uni-dimensional com qualquer força, dividindo o código em funções. Pode inspirar-se no ficheiro `functionofucntionwithpointers.zip` presente nos códigos da página da disciplina. Pretendemos também usar namespaces. Rescreva então o código anterior, na forma de novos códigos com as seguintes especificações.

Crie uma função `RuKuNew1` que implemente o passo do método de Runge-Kutta para EDOs de 2a ordem para a 2a Lei de Newton, no caso uni-dimensional. Os seus argumentos devem ser:

- a função em memória dinâmica `Fovm` que contém a expressão da segunda derivada $a = F/m$ em função dos argumentos (variável tempo, variável posição, variável velocidade - derivada da posição),
- a variável passo `h` da diferença finita ,
- a variável tempo (matematicamente é uma variável) inicial `t0`,
- a variável posição inicial (matematicamente é a função) `x0`,
- a variável velocidade inicial (matematicamente é a derivada da função) `v0`,

Observação: a função deveria retornar o tempo após um passo `t1`, a posição após um passo `x1` e a velocidade após um passo `v1`. No entanto as funções em C++ apenas retornam um valor, neste caso retorne `t1`. Para poder retornar mais valores crie um `namespace` por exemplo de nome `forRK`, onde define as variáveis `x1` e `v1`. A função `RuKuNew1` deve então usar esse `namespace`. Para além das aulas, pode ver exemplos no ficheiro `namespace.cpp` na página de FC.

Crie em seguida a função `Fovm` que nos dá a expressão de F/m vinda da Lei de Newton, em função dos argumentos (variável tempo, variável posição, variável velocidade - derivada da posição).

Observação: a função ainda necessita de conhecer parâmetros, que não queremos usar como argumentos afim de ficar geral. Aplicando ao caso particular em estudo, os parâmetros são vários, m, k, λ, F_0, w_f. Para tal vamos usar um novo `namespace` de nome `forF,`onde define estes parâmetros. A função `Fovm` deve então usar esse `namespace`.

A função `main` deve correr um ciclo `for` que resolve a EDO, e faça output para ficheiros dos resultados. Obviamente deve usar ambos os dois namespaces definidos para auxiliar as duas funçoes. Os dados iniciais e valores de eventuais parâmetros devem ser lidos dum outro ficheiro. Os valores do índice `i`, do tempo `t[i]`, da posição `x[i]` e da velocidade `v[i]` devem ficar guardados numa matriz em memória dinâmica enquanto o código corre. Deve fazer o output para um ficheiro.

Para ficar mais geral e robusto, guarde este código em quatro ficheiros independentes, tal como no ficheiro `functionofucntionwithpointers.zip` : um header .h, um .cpp para o Runge-Kutta, um .cpp para a força, e outro .cpp para a main.

Finalmente deve testar o código, reproduzindo os resultados de 1.a) no último caso particular que considerou (por exemplo $w_f = 3\,w_0$). Obviamente, deve produzir os gráficos respetivos.

2 EDOs acopladas : lei de Newton tri-dimensional

2.a Arrays

(4pt) Vamos agora estender a alínea 1.b) para o caso real de uma partícula no espaço-tempo tri-dimensional. Note então que:

- passamos a ter 3 EDOs de 2a ordem acopladas, uma para $x(t)$, outra para $y(t)$ e outra para $z(t)$, que dependem respetivamente de F_x/m, de F_y/m e de F_z/m;
- ou seja os argumentos e os valores que queremos retornar das funções incluem não uma componente mas sim as três componentes da posição $\mathbf{r}(t)$ e da velocidade $\mathbf{v}(t)$;
- bem como passamos também a ter três funções dinâmicas para a força, que contêm a expressão das componentes da segunda derivada $\mathbf{a}$ dada pela 2a Lei de Newton.

As funções que agora usamos denominam-se `RuKuNew3` e `Fovm3`.
Observação: uma forma de simplificar este problema é trabalharmos com arrays para a posição, a velocidade e a força. É trivial estender os namespaces de 1.b) para trabalharmos com arrays. Quanto à função `Fovm3` vamos também estende-la e ela passa a retornar um array (o que já era possível com a linguagem C). Um array não pode ser retornado diretamente, então retornamos um ponteiro para um array. A sintaxe pode ser,

```
double * Fovm3(double r[3], double v[3], double t) ) { // include the arguments
   static double  f[3]; // needs to be static
   ... // here we compute the three components f[0], f[1], f[3]
   ...
   return f;
}
```

(onde também temos arrays `double r[3]`, `double v[3]` como argumentos) e agora quando definimos a segunda função que usa esta função devemos usar um ponteiro, por exemplo

```
...
double RuKuNew3( double* (* Fovm3)( double[3], double[3], double), double h, ...
...
```

note bem que quando o argumento é um array isso deve ser explícito, escrevendo `double[3]` na definição do tipo. Quando usamos a primeira função, usamos de novo um ponteiro,

```
  ...
   double *p;
   p = Fovm3( ... ); // include the arguments
   ...
   ...=p[0]; // to use the first array element, etc
  ...
```

e finalmente, quando usa a primeira função, por exemplo na `main` , deve usar a sintaxe correcta para uma função criada com um ponteiro,

```
  ...
 double t1=  RuKuNew3( &Fovm3,h
 ...
```

Implemente então esta sintaxe no seu novo código, bem como a extensão das equações para incluírem três componentes.

Para aplicar a um caso particular, nesta alínea mantenha a força de 1.b) , mas atuando no eixo dos xOx apenas, deixando as componentes F_y e F_z nulas. Basta considerar o 1o caso particular para w_f. De novo escreva o output num ficheiro e com ele produza os gráficos em .pdf que mostrem que o código corre correctamente.

2.b Satélite

(4pt) Aplique ao caso dum satélite em órbita baixa, sujeito a duas forças, a atração universal de Newton

$$\mathbf{F}_{\text{gravitica}} = -G_{\text{newton}} \frac{m_{\text{satelite}} m_{\text{terra}}}{|\mathbf{r}|^3} \mathbf{r} \ ,$$

e a força de atrito sólido-ar

$$\mathbf{F}_{\text{atrito}} = -1/2 \rho(z - R_{\text{terra}}) \, A_{\text{satelite}} \, |\mathbf{v}| \mathbf{v} \ ,$$

onde a densidade atmosférica é exponencial. $\rho(z - R_{\text{terra}}) = \rho_0 \, \exp\left(-\frac{z - R_{\text{terra}}}{\bar{a}}\right)$.

Considere os seguintes parâmetros,
$G_{\text{newton}} = 6.674 \times 10^{-11}$ N (m/kg)^2, constante gravitacional,
$m_{\text{terra}} = 5.972 \times 10^{24}$ kg, massa da terra,
$R_{\text{terra}} = 6.371 \times 10^{6}$ m, raio médio da terra,
$m_{\text{satelite}} = 10^4$ Kg, massa do satélite,
$A_{\text{satelite}} = 10$ m^3, seção eficaz do satélite,
$\rho_0 = 1.225$ kg/m^3, densidade do ar,
$\bar{a} \simeq m_{\text{molecula}} g/(kT) \simeq 8000$ m, espessura média da atmosfera.
Deve ainda usar como parâmetros do código, o número máximo de iteradas `n_max` , o passo de iteração `h` e un número `n_jump` que define ao fim de quantos passos imprime valores no ficheiro de output. Note que vai ter de correr um número muito grande de iterações, pois `h` deve ser pequeno devido à elevada velocidade, e a órbita é grande, pelo que não vai conseguir fazer um gráfico se guardar o resultado de todas as iterações.

Considere as condições iniciais de uma órbita baixa, com altitude de 80 km apenas, por exemplo com coordenadas iniciais $\mathbf{r}_0 = (R_{\text{terra}} + 80\,\text{km}, 0, 0)$. Considere o caso de três velocidades iniciais diferentes $\mathbf{v}_0 = (0, v_{0y}, 0)$,
v_{0y}= 11.2 km/s (velocidade de escape),
v_{0y}= 9.7 km/s,
v_{0y}= 8.2 km/s (um pouco acima de 7.9 km/s, a velocidade da órbita circular).
Obviamente, deve ter atenção às unidades.

Deve colocar todos os parâmetros e dados iniciais em ficheiros de leitura. São demasiados parâmetros para os inserir à mão do teclado sempre que vai testar o programa.

Num ficheiro de escrita, para depois analisar, coloque todos os dados relevantes em colunas: o índice de iteração `n` o tempo t as 3 componentes do vetor posição $\mathbf{r}$, a altitude $|\mathbf{r}| - R_{\text{terra}}$, as 3 componentes do vetor velocidade $\mathbf{v}$ e o seu módulo $|\mathbf{v}|$.

Corra em cada caso algumas (poucas) órbitas e use o `mathematica` para representar 3 plots por caso:
um plot da altitude em função do tempo,
um plot do módulo da velocidade em função do tempo
um plot da trajetória no plano $x\,O\,y$, representando também no plot a Terra com um disco (comando `Disk` do `mathematica`), e a mesma escala nos dois eixos.

Comente se o satélite chega a orbitar a terra ou não, e se quando o faz cai na terra ao fim de poucas órbitas ou não, para cada condição inicial.

2.c Estrutura

(4pt) Optamos agora por criar uma estrutura o mais simples possível para trabalharmos com vetores tri-dimensionais.
Observação: reveja a estrutura `Point` definida no início da aula 7 (página3) ,

```
struct Point {
  int x, y;
  Point() { //construtor por defeito
    x = y = 0; // inicializa as variaveis
  }
  Point(int _x, int _y) { //construtor standard,
    x = _x;
    // conveniente para dar valores a parâmetros
    y = _y;
  }
};
```

que se pode estender para trabalhar com `double` e para ter três componentes `x` , `y` e `z`. Entretanto já criámos um código, `3Dvector.cpp` que está disponível na página da disciplina, onde encontra uma estrutura que já faz essa extensão, e ainda permite trabalhar com a notação `[i]` para os elementos de matriz (tem a vantagem de permitir o uso de ciclos).

Adapte o código de 2.b) para trabalhar com esta estrutura,em vez de trabalhar diretamente com arrays. Note que vai ter de fazer muitas substituições semelhantes, o que pode fazer depressa usando (com cuidado) o `Find and Replace` do seu editor de texto.

Aplique ao último caso particular de 2.b), produzindo de novo os ficheiros .pdf com o seu ficheiro de resultados calculados.

Comente numa breve frase se o código de 2.c) é mais simples de escrever, ou não, que o de 2.b). Dê ainda uma breve sugestão, como poderia simplifica-lo bastante mais usando outra funcionalidade das classes.