Scientific Computing using Octave: a working experience

Paola Gervasio

DICATAM, University of Brescia (Italy)
Personal teaching experience: Scientific Computing course. Lesson/laboratory for graduate students, Dept. of Computer Science and Engineering, University of Brescia


These books collect teaching experiences at Polimi (Milano), EPFL (Lausanne) and UniBS (Brescia).
Personal teaching experience

- **Scientific Computing (SC) course**
  - 1st year of master degree in “Computer Science and Engineering”
  - class of about 50-60 students
  - lessons: 40 hours
  - exercises with octave/matlab: 40 hours

- **Students’ background:**
  - Programming languages: C, C++, Java, Html
  - web design
  - software engineering
  - operating systems and computing infrastructures
  - ....

This course offers the first approach to both numerical analysis and scientific computing
“Initialization step”
- basic instructions in matlab/octave
- machine arithmetic
- errors in computations
- costs of computations

Solving nonlinear equations
Approximation of data and functions
Linear systems
Numerical integration and differentiation
Ordinary differential equations
Partial differential equations (very simple problems with FD)
Motivations in using Octave

1. to get in touch with machine arithmetic
2. to better understand basic concepts of numerical analysis: errors, stability, convergence, accuracy, ...
3. to take advantage of built-in functions
4. to compare methods by measuring time effort, accuracy, reliability of the results
5. to learn cost-effective programming techniques
6. to support theoretical explanations by graphics

\[ \frac{|x - f_i(x)|}{|x|} \leq \frac{1}{2} \epsilon_M \]
Octave support

- **built-in functions:**
  - linspace, meshgrid, plot, mesh, surf, contour,
  - det, rank, eig, cond, norm,
  - \, lu, chol, luinc, qr, pcg, bicgstab,
  - fzero, fsolve,
  - polyfit, polyval, interp1, interp2, spline, mkpp, ppval,
  - trapz, quad, quadl,
  - ode23, ode45,

- **programming language**

- **hand made functions:**
  - Newton, Broyden, fixed point iterations,
  - adaptive Simpson rule,
  - LU factorization without pivoting, Gaussian elimination, Jacobi, Gauss-Seidel, Richardson,
  - Euler, Crank-Nicolson, AB-AM predictor correctors, fixed step RK methods,
  - 2D finite difference approximation of Laplace and heat equation.
Exercise 1. Load the matrix $A$ and the r.h.s. $b$ stored in ex1.mat. Analyze both structure and properties of the matrix and then solve $Ax = b$ with the most appropriate direct method among them presented during the course.

Solution

Student:
octave:5> load ex1
octave:6> whos
Variables in the current scope:
        Attr Name      Size             Bytes  Class      Attributes
        ==== ====      ====             =====  =====
A     100x100          80000  double          
b     100x1          80000  double          

Total is 10100 elements using 80800 bytes
octave:7> d=det(A)
d = 0

The matrix is singular!!!!

It is not possible, you are wrong, compute the rank, please

octave:8> r=rank(A)
r = 100

Surprise... What happens?

Let us compute the eigenvalues of $A$
Sub-exercise: Load the matrix $A \in \mathbb{R}^{100 \times 100}$ stored in ex1.mat. Explain why it results $\det(A)=0$ while $\text{rank}(A)=100$. 

octave:9> v=eig(A)

v =
   1.0000e-02
   9.1116e-03
   8.3022e-03
   .
   .
   1.2045e-06
   1.3219e-06
   1.4508e-06
   1.5923e-06
   1.7475e-06
   1.9179e-06

All eigenvalues are strictly positive....

OK: let us consider this sub-exercise
Solution

Let \( \lambda_k \) denote the eigenvalues of \( A \).

For \( n = 1, \ldots, 100 \), compute and print \( p_n = \prod_{k=1}^{n} \lambda_k \).

```
load ex1; v=eig(A);
p=v(1);
for n=2:100
  p=p*v(n);
  fprintf('product of first %d eigenval = %13.6e \n',n, p)
end
```

```
product of first 2 eigenval = 9.111628e-05
product of first 10 eigenval = 1.519911e-22
product of first 70 eigenval = 2.656088e-238
product of first 86 eigenval = 2.104720e-320
product of first 87 eigenval = 0.000000e+00
```

\( p_{86} < \text{realmin}?? \)

\( p_{87} = 0?? \)

This is a way of exploring the floating-point set \( \mathbb{F}(2, 53, -1021, 1024) \), realmin and realmax, underflow and overflow, normal and denormal floating-point numbers.
Machine arithmetic (continued)

Exercise 2. We know that \( \lim_{n \to \infty} \left(1 + \frac{1}{n}\right)^n = e \).

By evaluating and plotting \( a_n = \left(1 + \frac{1}{n}\right)^n \) (for \( n = 1, \ldots, 10^{20} \)), the graph on the right is produced. Explain the behaviour of the sequence \( a_n \).

Solution

```matlab
N=1.e20; n=1; an=[]; nn=[];
while n< N 
a1=(1+1/n)^n;
an=[an;a1]; nn=[nn;n]; n=n*5;
end
semilogx(nn,an,'.','Markersize',24);
hold on
semilogx([1,N],[exp(1),exp(1)],'r--', 'Linewidth',3);
```

Machine precision in practice:
when \( n \geq 9.3260e+15 \Rightarrow \frac{1}{n} \lesssim 1.0723e-16 < \epsilon_M \)
and then \( 1 + \frac{1}{n} = 1 \) in \( \mathbb{F} \).
The aim is to experience benefits of the pivoting in solving linear systems by LU factorization, but at the same time we realize high performance of built-in functions. Call the `lu` function (which uses pivoting by default) and write `lu_nopiv` function (without pivoting).

```matlab
function A=lu_nopiv(A)
%A=lu_nopiv(A)
[n,m]=size(A);
if n~= m
    disp('non-square matrix'); return
end
for k=1:n
    if A(k,k)==0
        disp('Singular submatrix'); return
    end
    for i=k+1:n
        A(i,k)=A(i,k)/A(k,k);
        for j=k+1:n
            A(i,j)=A(i,j)-A(i,k)*A(k,j);
        end
    end
end
end
```

```matlab
t1=cputime;
A1=lu_nopiv(A);
L1=tril(A1,-1)+eye(n); U1=triu(A1);
t2=cputime;
z=L1; x=U1;
err=norm(x-xex)/norm(xex);
t3=cputime; [L,U,P]=lu(A); t4=cputime;
z=L(P*b); x=U;z;
err=norm(x-xex)/norm(xex);
```

My function is low performing!!! ➞ vector instructions

```
Output
lu_nopiv cputime= 5.08e+00, err= 1.783394e-12
lu cputime= 2.00e-03, err= 1.130656e-13
```

19.1 Basic Vectorization

To a very good first approximation, the goal in vectorization is to write code that avoids loops and uses whole-array operations. As a trivial example, consider

```octave
def i = 1:n
  for j = 1:m
    c(i,j) = a(i,j) + b(1,j);
  endfor
endfor
```

cmpared to the much simpler

```octave
c = a + b;
```

This isn't merely easier to write; it is also internally much easier to optimize. Octave delegates this operation to an underlying implementation which, among other optimizations, may use special vector hardware instructions or could conceivably even perform the additions in parallel. In general, if the code is vectorized, the underlying implementation has more freedom about the assumptions it can make in order to achieve faster execution.

This is especially important for loops with "cheap" bodies. Often it suffices to vectorize just the innermost loop to get acceptable performance. A general rule of thumb is that the "order" of the vectorized body should be greater or equal to the "order" of the enclosing loop.
First attempt: modify the main loop in \texttt{lu\_nopiv}

```matlab
function A=lu_nopiv_v(A)
% A=lu_nopiv_v(A)
[n,m]=size(A);
if n~= m
    disp('non-square matrix')
    return
end
for k=1:n
    if A(k,k)==0
        disp('Singular submatrix, pivoting is required')
        return
    end
    A(k+1:n,k)=A(k+1:n,k)/A(k,k);
    A(k+1:n,k+1:n)=A(k+1:n,k+1:n)-A(k+1:n,k)*A(k,k+1:n);
end
```

New output

- scalar function cputime = 4.97e+00, err = 3.907288e-13
- vector function cputime = 1.40e-02, err = 3.907288e-13
- built-in function cputime = 3.00e-03, err = 5.015366e-14

The new code is not the best one, but it is better than the scalar one! Try to improve performance by swapping loops.
Spherical pendulum

The motion of a point $\mathbf{x}(t) = (x_1(t), x_2(t), x_3(t))^T$ with mass $m$ subject to the gravity force $\mathbf{F} = (0, 0, -gm)^T$ (with $g = 9.8 \text{ m/s}^2$) and constrained to move on the spherical surface of equation $\Phi(\mathbf{x}) = x_1^2 + x_2^2 + x_3^2 - 1 = 0$ is described by the following system of ordinary differential equations

$$\dot{\mathbf{x}} = \frac{1}{m} \left( \mathbf{F} - \frac{m \mathbf{x}^T \mathbf{H} \dot{\mathbf{x}} + \mathbf{x} + \nabla \Phi^T \mathbf{F}}{|\nabla \Phi|^2} \nabla \Phi \right)$$

for $t > 0$.

To numerically solve the system, let us transform it into a system of differential equations of order 1 in the new variable $\mathbf{y} = [x_1, x_2, x_3, \dot{x}_1, \dot{x}_2, \dot{x}_3]$, and apply Euler, Runge-Kutta (etc...) methods to the system

$$\begin{cases} \mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)), & t \in (t_0, T] \\ \mathbf{y}(t_0) = \mathbf{y}_0 \end{cases}$$

When stability is satisfied, we can have an idea of the accuracy by noticing that the solution satisfies $r(\mathbf{y}) \equiv |y_1^2 + y_2^2 + y_3^2 - 1| = 0$ and by consequently measuring the maximal value of the residual $r(\mathbf{y}_n)$ when $n$ varies, $\mathbf{y}_n$ being the approximation of the exact solution generated at time $t_n$. 
function \([t,u]=\text{feuler}(\text{odefun},tspan,y0,...\)
\(\text{Nh}, \text{varargin})\)
\[ h=(tspan(2)-tspan(1))/\text{Nh}; \]
\[ y=y0(:); \ w=y; \ u=y.'; \]
\[ \text{tt}=\text{linspace}(tspan(1),tspan(2),\text{Nh}+1); \]
\[ \text{for } t = \text{tt}(1:end-1) \]
\[ \quad w=w+h*\text{odefun}(t,w,\text{varargin}{:}); \]
\[ \quad u = [u; \ w.']; \]
\[ \text{end} \]
\[ t=\text{tt}'; \]

\begin{verbatim}
function [f]=fpendulum(t,y)
f=zeros(size(y)); H=2*eye(3);
xpunto=zeros(3,1);
xpunto=y(4:6);
mass=1; F=[0;0;-mass*9.8];
G=zeros(3,1); G=2*y(1:3);
lambda=(mass*xpunto'*H*xpunto+F'*G)/(G'*G);
f(1:3)=y(4:6);
for k=1:3;
f(k+3)=(F(k)-lambda*G(k))/mass;
end
\end{verbatim}

In the 1st run the solution blows up. Accuracy
and cputime:

<table>
<thead>
<tr>
<th>nt</th>
<th>residual</th>
<th>cputime</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>1.0578</td>
<td>2.69</td>
</tr>
<tr>
<td>100000</td>
<td>0.1111</td>
<td>229.31</td>
</tr>
</tbody>
</table>

Theory confirmed. Euler is time consuming
for small \(h\). Residuals are large, then more
accurate schemes are right.
We consider 4th-order RK schemes with both constant steplengths (hand made function `rk4`) and adaptive steplengths (`ode45` function of `odepkg`). We compare accuracy and computational costs:

```matlab
y0=[0,1,0,8,0,1.2]; tspan=[0,25];
[t1,u1]=ode45(@fpendulum,tspan,y0);
r1=abs(u1(end,1)^2+u1(end,2)^2+u1(end,3)^2-1)
ode45 requires 648 steps and yields r1 = 3.7866e-04

[t2,u2]=rk4(@fpendulum,tspan,y0,648);
r2=abs(u2(end,1)^2+u2(end,2)^2+u2(end,3)^2-1);
rk4 yields r2 = 0.10906.
rk4 requires 3000 steps to yield residual \( \approx 10^{-4} \)

[t3,u3]=rk4(@fpendulum,tspan,y0,3000);
r3=abs(u3(end,1)^2+u3(end,2)^2+u3(end,3)^2-1);
r3 = 2.3444e-04
```

<table>
<thead>
<tr>
<th>scheme (nt)</th>
<th>residual</th>
<th>cputime</th>
</tr>
</thead>
<tbody>
<tr>
<td>ode45 (648)</td>
<td>3.7866e-04</td>
<td>0.98</td>
</tr>
<tr>
<td>rk4 (648)</td>
<td>0.10906</td>
<td>0.45</td>
</tr>
<tr>
<td>rk4 (3000)</td>
<td>2.3444e-04</td>
<td>2.14</td>
</tr>
<tr>
<td>rk4 (1500)</td>
<td>3.7459e-03</td>
<td>1.07</td>
</tr>
</tbody>
</table>

Paola Gervasio
Octave and Matlab are very useful in showing the behaviour of numerical methods. This is the case of numerical optimization.
Graphic support to theoretical lessons

Graphics functions provide fundamental support when teaching: to analyse output of scientific computing, but also to create movies for the lessons (mathematical analysis in this case).
A few (known) suggestions

- Some times the help does not report which algorithm is implemented inside a function.
- To make immediate the use of octave for new users, examples inside the help would be very useful.
- Compatibility with Matlab is very appreciated on basic graphic instructions as well as for string/functions manipulation.
- On linux platform (fedora), installation of the latest release of Octave by source file is not always easy, it depends on OS, installed sw, ....
A remark about the command $\text{view}$

The command $\text{octave:18> view([79,21])}$ yields

![Figure 1](image)

It seems that $\text{view([az,el])}$ actually yields view position $= [90-\text{el}, \text{az}]$
Thanks!