Aalto SCIp **Heikki Apiola, Juha Kuortti**

Matlab 2

For credits, send in at least 3 problems to us:
heikki.apiola@aalto.fi, juha.kuortti@aalto.fi
Problems 1,2,3 use parallel toolbox, problems 4,5 don't require it.

Dead-line for credits to be discussed on lecture 23.11.

1. Experiments with `spmd`-way of numerical integration in Triton.

```
ssh -X scip@triton.aalto.fi
passw: Given in lectures
mkdir mynamed  % Where, perhaps under course dir
   Slight problem with common project (?)
cd mynamed
module load matlab
```

cp scip2016/2016_2_syksySCI/Lectures/spmd_numint.m .
matlab&

   (a) Open *spmd_numint.m* in Matlab, experiment with some nlabs-values and enjoy!

   (b) Plot the Matlab-function `humps` (`help humps`). Run the spmd-block using `humps`

   (c) Take a lower degree integration method: `help trapzd`. It integrates summing up trapezoids defined by x and y-vectors. Write a function:

```
function  z  = trapzfun(f,a,b,n)
% z  = trapzfun(f,a,b,n) returns trapez-sum defined by
%      vectors x and y=f(x), x=linspace(a,b,n)
...
```

   (d) Now run spmd using your `trapzfun`-function using several nlabs-values and observe the affects to accuracy. Remember: `format long`

   Write a script with explanations and experiences. Publish it into pdf:
   `>> publish 'spmdnumint.m'` (Remember: Give this in command window, not in the script-file -> infinite loop).

2. Here's the beginning part of the "darts"-simulation-estimation for π in Matlab 1-course.

   (a) Plot the unit square : $-1 \leq x \leq 1, -1 \leq y \leq 1$ and the unit circle inside it.

```
t=linspace(0,2*pi);
x=cos(t);y=sin(t);
plot(x,y,[1 1 -1 -1 1],[-1 1 1 -1 -1]);
axis([-1.5 1.5 -1.5 1.5])
axis square
```

(b) Generate random points, uniformly distributed on the unit square $-1 < x <, 1 < y < 1$. Find an approximation for π by counting the ratio of the number of points inside the circle with the total number of points generated.

Hint: Generate two random vectors x and y. Find a suitable condition for logical indexing to pick the "inside-poins".
Note: You can use arithmetic (sum) to a logical vector. Try to avoid loops.

Matlab 2-exercise:

Do the simulation in parallel in at least one of two ways:

1. spmd
2. parfor

```
nlabs=2;  % Begin
nlabs=16; % Triton (vary this)
% Recommendation: Try Triton also
parpool(nlabs)
spmd
N=100000;     % Vary
X = something random
Y = something random
% Condition for hitting inside disk:
in = ...;
Nhits = ...;
piestimlabs= ...  % Each worker's result
end
```

Use `gplus` to compute the total number of hits.

If you want to try parfor, you must break the elegant vectorization and write a "traditional" loop. But you get material for efficiency comparisons. (This is "voluntary".)

Please observe: Don't expect miracles in accuracy, Monte Carlo is a low-accuracy method.

3. *Van der Pol equation* is an equation describing self-sustaining oscillations in which energy is fed into small oscillations and removed from large oscillations.

$$y'' - \mu(1 - y^2)y' + y = 0$$

(a) Use Matlab-solver `ode45` for solving the equation for some value, say $\mu = 1.5$.
For this you have to transform the equation into a systen of first order equations:

$$\begin{cases} y_1' = y_2 \\ y_2' = \mu(1 - y_1^2)y_2 - y_1 \end{cases}$$

Let me give the Matlab code:

```
function dydt = vdpode(t,y,mu)
% Vanderpol equation
%
dydt = [y(2); mu*(1-y(1)^2)*y(2)-y(1)];
```

The trick is that in the script or function where you call the ode solver (`ode45`) you can define and use a function handle like this:

```
mu=1.5;
odefun=@(t,y)=vdpode(t,y,mu)
```

(Each time you change `mu` you have to re-enter the function definition line to get the updated `mu` in the definition. If these lines are included in a fuction or script, then it happens automatically, of course.)

Test your function first by something like:

`[T,Y]=ode45(odefun,[0 tmax],[0 1]);` (see `help ode45`.)

Use suitable `tmax`. (`[0 1]` are initial conditions, you may vary.)

(b) Create job and tasks: Instead of a script you need to write a "coverfunction" `vdpSolver` whose parameter list includes `mu` also. For simplicity, just fix the initial conditions and `tmax` for instance as `[0 1]` and 25.

Then you can create a job and tasks:

```
job=createJob(...);

task1 = createTask(job,@vdpSolver,2,{0.5});
task2 = createTask(job,@vdpSolver,2,{1});
task3 = createTask(job,@vdpSolver,2,{1.5});
%% Submit:
submit(job)
wait(job)
results =fetchOutputs(job);
```

Then you can plot solutions, phase-planes, see the affect of mu.

We will give more hints and comparison to a similar problem with some more parameters on the remaining lectures.

4. A simple scrambling circuit for voice communications works as follows. Consider the frequency band from 0 to 4 kHz. It is a known fact that the overwhelming majority of the power spectrum of human voice is concentrated in this frequency band. One way of scrambling this frequency band is to subdivide it into 4 equal sub-bands and interchange the sub-bands according to some pre-determined key. For example, let sub-band A correspond to frequencies between 0 and 1 kHz. Then, sub-band B corresponds to frequencies between 1 and 2 kHz, sub-band C corresponds to frequencies between 2 and 3 kHz, and sub-band D corresponds to frequencies between 3 and 4 kHz. The original order of the sub-bands is ABCD. A simple scrambling technique is to interchange this order, i.e.

reorder the sub-bands to BCDA or DCBA or CABD or any other pre-determined order. Call the resulting signal the scrambled signal. This scrambled signal is not comprehensible unless you know the key and can rearrange the sub-bands back into the original order.

The goal of this problem is for you to design a MATLAB program that will descramble a given voice signal. You may obtain the scrambled signal ('scramble.wav') from the course web site. To transform the voice signal into frequency domain, you should use the MATLAB command `fft`; to rearrange the bands you should take the first half of the transform (beware of off-by-one), and rearrange it using the key given below; after that, use `ifft` to transform the signal back to time domain. You can play any sound clip in MATLAB using the command `sound`; remember to provide the sampling rate, and don't do it during class (`sound` doesn't respect Ctrl-c).

It has been scrambled in the above-described fashion by rearranging the bands ABCD into CBDA. Descramble this signal and transcribe the first and last 5 words. Turn in your code as well as your transcription.

This assignment and the sound clip are adapted from materials in Stanford university course EE261 distributed under Creative Commons BY-NC-SA 4.0 license.

5. Let c and z_0 be complex numbers. We define the following recursion:

$$z_n = z_{n-1}^2 + c$$

This is a dynamical system known as a quadratic map. Given different choices for parameter c and the initial value z_0 the recursion leads to a sequence of complex numbers $z_1, z_2, \ldots$ known as the orbit of z_0. This dynamical system is highly *chaotic*, meaning that depending on the selected c and z_0, a huge number of different orbit patterns are possible.

Suppose that we fix the parameter c. In such cases, most choices of z_0 tend towards infinity (i.e. $|z_n| \to \infty$ as $n \to \infty$). For some z_0 (this depends a little on c as well), however, the orbit is stable, meaning that it goes into periodic loop; and finally there are some orbits, that seem to do neither, dancing around the complex space apparently at random.

In this assignment, your task is to you write a MATLAB script that visualizes a slightly different set, called the filled-in Julia set (or Prisoner Set), denoted K_c , which is the set of all z_0 with orbits which do not tend towards infinity. The "normal" Julia set would be the edges of of K_c.

a) It is known that if the modulus of z_n (i.e. $|z_n|$) becomes larger than 2 for any n, the sequence will tend to infinity. The value of n for which this becomes true is called the 'escape velocity' of a particular z_0. Write a function that returns the escape velocity of given z_0 and c. Note you cannot test the recursion for all n: but rather you should select an upper bound N, so that if $|z_n| < 2 \, \forall n < N$, the function should return N. This allows you to avoid infinite loops.

b) Then write a function that takes $c, z_{\max}$ and N as arguments. The function will define a square in complex plane of complex numbers with real part between $-z_{max}$

and z_{max}, and imaginary part between $-z_{max}$ and z_{max}, and discretise it into a 500×500 grid. It will then compute the escape velocity of every element in the grid using the function you wrote previously, and the parameters c and N. Save the escape velocities to a matrix M; remember to preallocate.

c) Visualize your fractal using `imagesc(M)`. You may also want to try `imagesc(atan(0.1*M))`.