

```

> restart;
> Digits:=15;
> with(LinearAlgebra):
> t11:=time():
> t12:=time[real]():

```

Code was updated on 05/28/22 to call the sparse jacobian in vector format. This enables compiled call for both the residues and the sparse entries of the Jacobian. The structure and 'sym' part of the UMFPACK linesolver are called only once.

### Code is released under MIT license

Code updated on 05/02/22 to minimize reduce overhead calls for LinearSolve (based on inputs from acer).

It directly calls MatVecSolve in UMFPACK. Many procedures are autocompiled.

NN is the number of node points (elements) in the X and MM is the number of elements in the Y direction. delta is the applied current density at the top (Y =1). tf is the final time for simulation. vel is the velocity constant v in the paper. ki0 is the scaled exchange current density k in the paper. This code can be run for positive values of delta. This simulates plating. At the end of simulation, changing delta to negative values and rerunning the code will automatically use the geometry at the end of plating. Ydatstore stores the geometry at every point in time. Phiaveadd stores the total liquid phase in the domain at any point in time.

Users can change NN, delta, tf, vel, ki0, MM just in this line and choose Edit execute worksheet to run for different design parameters.

Users can modify the call for y0proc for choosing different models.

Users can modify the call for HF to run first-order upwind, ENO2 or WENO3 methods. NN and MM should be even numbers.

```

> NN:=100;MM:=NN;delta:=0.1;vel:=1.0;ki0:=1.0;tf:=2.0;
      NN := 100
      MM := 100
      δ := 0.1
      vel := 1.0
      ki0 := 1.0
      tf := 2.0

```

(1)

```

> gc();
> N:=NN+2:
> M:=MM+2:
> h:=1.0/NN:
> ntot:=N*M;
      ntot := 10404

```

(2)

Initial geometry, Model 1, semicircle in a square

```

> y0proc1:=proc(NN,MM,Y00)
  local i,j,xx,yy,N,h,w,M,Ny,rf,f,ff,rr;
> N:=NN+2:h:=1.0/NN:w:=h/2:
> M:=MM+2;Ny:=MM;

```

```

> for i from 2 to N-1 do for j from 2 to M-1 do
  xx:=-0+(i-1/2-1)*h:yy:=-0+(j-1/2-1)*h:
  rr:=xx^2+yy^2;
  Y00[i,j]:=max(1e-9,0.5+0.5*tanh((sqrt(rr)-0.3)/w/sqrt(2.0))): 
  od:od:
  for i from 1 to N do
  Y00[i,1]:=Y00[i,2]:
  Y00[i,M]:=Y00[i,M-1]:
  od:
  for j from 1 to M do
  Y00[1,j]:=Y00[2,j]:
  Y00[N,j]:=Y00[N-1,j]:
  od:
end proc:
```

Model 2, square in a square

```

> y0proc2:=proc(NN,MM,Y00) # square inside a circle, Model 2
  local i,j,xx,yy,N,h,w,M,Ny,rf,f,ff,rr;
> N:=NN+2:h:=1.0/NN:w:=h/2:
> M:=MM+2:Ny:=MM;
> for i from 2 to N-1 do for j from 2 to M-1 do
  xx:=-0+(i-1/2-1)*h:yy:=-0+(j-1/2-1)*h:
  if xx <=0.3 and yy<=0.3 then Y00[i,j]:=1e-9; else Y00[i,j]:=1.0:end:
  od:od:
  for i from 1 to N do
  Y00[i,1]:=Y00[i,2]:
  Y00[i,M]:=Y00[i,M-1]:
  od:
  for j from 1 to M do
  Y00[1,j]:=Y00[2,j]:
  Y00[N,j]:=Y00[N-1,j]:
  od:
end proc:
```

Initial geometry, Model 3, electrodeposition problem trenches and via

```

> y0proc3:=proc(NN,MM,Y00)
  local i,j,xx,yy,N,h,w,M,Ny,rf,f,ff,rr;
> N:=NN+2:h:=1.0/NN:w:=h/2:
> M:=MM+2:Ny:=MM;
> for i from 2 to N-1 do for j from 2 to M-1 do
  xx:=-0+(i-1/2-1)*h:yy:=-0+(j-1/2-1)*h:
```

```

if abs(xx-0.5) >0.2 and yy<=0.5 then Y00[i,j]:=1e-9; else Y00[i,
j]:=1.0:end:
od:od:
for i from 1 to N do
Y00[i,1]:=Y00[i,2]:
Y00[i,M]:=Y00[i,M-1]:
od:
for j from 1 to M do
Y00[1,j]:=Y00[2,j]:
Y00[N,j]:=Y00[N-1,j]:
od:
end proc:

```

Initial geometry, Model 4, Gaussian Seed at the bottom

```

> y0proc4:=proc(NN,MM,Y00)
local i,j,xx,yy,N,h,w,M,Ny,rf,f,ff,rr;
> N:=NN+2:h:=1.0/NN:w:=h/2:
> M:=MM+2;Ny:=MM;
> for i from 2 to N-1 do for j from 2 to M-1 do
xx:=-0+(i-1/2-1)*h:yy:=-0+(j-1/2-1)*h:
rr:=0.1+0.1*exp(-500.* (xx-0.5)^2);
Y00[i,j]:=max(1e-9,0.5+0.5*tanh((yy-rr)/w/sqrt(2.0))):
od:od:
for i from 1 to N do
Y00[i,1]:=Y00[i,2]:
Y00[i,M]:=Y00[i,M-1]:
od:
for j from 1 to M do
Y00[1,j]:=Y00[2,j]:
Y00[N,j]:=Y00[N-1,j]:
od:
end proc:

```

```

> y0proc:=evalf(y0proc1):#choose different models using y0proc2,
etc.
> Y00:=Matrix(1..N,1..M,datatype=float[8]):
> evalhf(y0proc(NN,MM,Y00)):
> if delta<0 then read("Y0data.m"):end:
> if delta<0 then read("tdata.m"):end:

> p0:=plots:-surfdata(Y00,-h/2..NN*h+h/2,-h/2..MM*h+h/2,dimension=

```

```
2,style=surface,colorscheme = ["Red", "Green", "Blue"]):
```

Next, boundary conditiosn at X = 0, X=1, Y = 0, Y = 1 are specified below, but these equations are not used and optimally coded inside the procedure Eqs11.

```
> eq:=Array(1..N,1..M):
> for i from 1 to 1 do for j from 1 to M do eq[i,j]:=-Phi[i,j]+Phi[i+1,j]:od:od:
> for i from N to N do for j from 1 to M do eq[i,j]:= Phi[i-1,j]-Phi[i,j]:od:od:
> for i from 1 to N do for j from 1 to 1 do eq[i,j]:= Phi[i,j+1]-Phi[i,j]:od:od:
> for i from 1 to N do for j from M to M do eq[i,j]:= Phi[i,j-1]-Phi[i,j]+delta*h:od:od:
```

Residues at different points in X and Y are coded in the Eqs11 procedure. Y0 is the input phase-field parameter (2D Matrix). The potential y is a vector to expedite the calculation of residue.

```
> Eqs11:=proc(Y0::Matrix(datatype=float[8]),y::Vector(datatype=float[8]),delta::float,ki0::float,ff::Vector(datatype=float[8]),N::integer,M::integer)
local i::integer,j::integer,i1::integer,h::float[8];
option optimize, autocompile;
h:=1.0/(N-2):
for j from 1 to M do
for i from 1 to N do
i1:=i+(j-1)*N:
if i>1 and i <N and j>1 and j<M then
ff[i1]:=
(Y0[i,j]+Y0[i,j+1])*(y[i1+N]-y[i1])
-(Y0[i,j]+Y0[i,j-1])*(y[i1]-y[i1-N])
+(Y0[i+1,j]+Y0[i,j])*(y[i1+1]-y[i1])
-(Y0[i,j]+Y0[i-1,j])*(y[i1]-y[i1-1])
-y[i1]*h*(1e-24+(Y0[i+1,j]-Y0[i-1,j])^2+(Y0[i,j+1]-Y0[i,j-1])^2)
^(1/2):
end:
if i =1 and j =1 then ff[i1]:=-y[i1]:end:
if i =N and j =1 then ff[i1]:=-y[i1]:end:
if i =1 and j =M then ff[i1]:=-y[i1]:end:
if i =N and j =M then ff[i1]:=-y[i1]:end:

if i >1 and i<N and j =1 then ff[i1]:=-y[i1]+y[i1+N]:end:
if i >1 and i<N and j =M then ff[i1]:=-y[i1]+y[i1-N]+delta*
h:end:
```

```

if j >1 and j<M and i =1 then ff[i1]:=-y[i1]+y[i1+1]:end:
if j >1 and j<M and i =N then ff[i1]:=-y[i1]+y[i1-1]:end:

od:
od:
end proc:

> jsproc:=proc(Y00::Matrix(datatype=float[8]),y::Vector(datatype=
float[8]),delta::float,ki0::float,jss::Vector(datatype=float[8]),
N::integer,M::integer)
local count::integer,i::integer,j::integer,cent::float[8],
top::float[8],bot::float[8],left::float[8], right::float[8],
h::float[8];
option optimize, autocompile;
h:=1.0/(N-2):
count:=0:for j from 1 to M do
for i from 1 to N do
if i>1 and i<N and j>1 and j<M then
count:=count+1:
if j = 2 then bot:=1.0: else bot:=Y00[i,j]+Y00[i,j-1]:end:
jss[count]:=bot:

count:=count+1:
if i = 2 then left:=1.0; else left:=Y00[i,j]+Y00[i-1,j]:end:
jss[count]:=left:

count:=count+1:
cent:=-4*Y00[i,j]-Y00[i,j+1]-Y00[i,j-1]-Y00[i+1,j]
-Y00[i-1,j]-ki0*h*(1e-24+(Y00[i+1,j]-Y00[i-1,j])^2
+(Y00[i,j+1]-Y00[i,j-1])^2)^(1/2):
jss[count]:=cent:#print(cent);

count:=count+1:
if i = N-1 then
right:=1.0; else right:=Y00[i,j]+Y00[i+1,j]:end:
jss[count]:=right:

count:=count+1:
if j=M-1 then top:=1.0: else top:=Y00[i,j]+Y00[i,j+1]:end:
jss[count]:=top:
end:

if i =1 and j =1 then count:=count+1: cent:=-1.0: jss[count]:=
```

```

cent: end:
  if i =N and j =1 then count:=count+1: cent:=-1.0: jss[count]:=cent: end:
  if i =1 and j =M then count:=count+1: cent:=-1.0: jss[count]:=cent: end:
  if i =N and j =M then count:=count+1: cent:=-1.0: jss[count]:=cent: end:

  if i >1 and i<N and j =1 then
  count:=count+1:cent:=-1.0: jss[count]:=cent:
  count:=count+1:top:=Y00[i,j]+Y00[i,j+1]:jss[count]:=top: end:

  if i >1 and i<N and j =M then
  count:=count+1:bot:=Y00[i,j]+Y00[i,j-1]: jss[count]:=bot:
  count:=count+1:cent:=-1.0: jss[count]:=cent: end:

  if j >1 and j<M and i =1 then
  count:=count+1:cent:=-1.0: jss[count]:=cent:
  count:=count+1:right:=Y00[i+1,j]+Y00[i,j]: jss[count]:=right: end:

  if j >1 and j<M and i =N then
  count:=count+1:left:=Y00[i,j]+Y00[i-1,j]: jss[count]:=left:
  count:=count+1:cent:=-1.0: jss[count]:=cent: end:

od:
od:
end proc:

> ApAiproc:=proc(Api::Vector(datatype=integer[4]),Aii::Vector
(datatype=integer[4]),N::integer,M::integer)
local count::integer[8],i::integer,j::integer,ii::integer:
option optimize, autocompile;
count:=0:
for j from 1 to M do
for i from 1 to N do
ii:=i+(j-1)*N:
if i>1 and i<N and j>1 and j<M then
count:=count+1:
Aii[count]:=i+(j-1)*N-1-N:Api[ii]:=count-1:
count:=count+1:

```

```

Aii[count]:=i+(j-1)*N-1-1:

count:=count+1:
Aii[count]:=i+(j-1)*N-1:#print(cent) ;

count:=count+1:
Aii[count]:=i+(j-1)*N-1+1:

count:=count+1:
Aii[count]:=i+(j-1)*N-1+N:
end:

if i =1 and j =1 then count:=count+1: Aii[count]:=i+(j-1)*
N-1:Api[ii]:=count-1:end:
if i =N and j =1 then count:=count+1: Aii[count]:=i+(j-1)*
N-1:Api[ii]:=count-1:end:
if i =1 and j =M then count:=count+1: Aii[count]:=i+(j-1)*
N-1:Api[ii]:=count-1:end:
if i =N and j =M then count:=count+1: Aii[count]:=i+(j-1)*
N-1:Api[ii]:=count-1:end:

if i >1 and i<N and j =1 then
count:=count+1:Aii[count]:=i+(j-1)*N-1:Api[ii]:=count-1:
count:=count+1:Aii[count]:=i+(j-1)*N-1+N:end:

if i >1 and i<N and j =M then
count:=count+1:Aii[count]:=i+(j-1)*N-1-N:Api[ii]:=count-1:
count:=count+1:Aii[count]:=i+(j-1)*N-1:end:

if j >1 and j<M and i =1 then
count:=count+1:Aii[count]:=i+(j-1)*N-1:Api[ii]:=count-1:
count:=count+1:Aii[count]:=i+(j-1)*N-1+1:end:

if j >1 and j<M and i =N then
count:=count+1:Aii[count]:=i+(j-1)*N-1-1:Api[ii]:=count-1:
count:=count+1:Aii[count]:=i+(j-1)*N-1:end:

od:
od:
ii:=ii+1:
Api[ii]:=Api[ii-1]+1:
end proc:

```

```
> #Compiler:-Compile(ApAiproc);
```

The Jacobian for the residues is coded in the procedure Jac1. Note that in this code the full matrix for the Jacobian is not used and Jac1 is not used. Jac1 is provided only to help the readers understand the procedure jsproc (nonzero values of Jac1) and the sparse storage format

```
> Jac1:=proc(Y0::Matrix(datatype=float[8]),y::Vector(datatype=float[8]),delta::float,ki0::float,j00::Matrix(datatype=float[8],storage=sparse),N::integer,M::integer)
local i::integer,j::integer,i1::integer,h::float[8];
#option optimize, autocompile;
option optimize;
h:=1.0/(N-2);

for j from 2 to M-1 do
for i from 2 to N-1 do
  i1:=i+(j-1)*N;
  j00[i1,i1]:=-4*Y0[i,j]-Y0[i,j+1]-Y0[i,j-1]-Y0[i+1,j]
    -Y0[i-1,j]-ki0*h*(1e-24+(Y0[i+1,j]-Y0[i-1,j])^2
    +(Y0[i,j+1]-Y0[i,j-1])^2)^(1/2):
  j00[i1,i1+1]:=Y0[i+1,j]+Y0[i,j]:j00[i1,i1-1]:=Y0[i-1,j]+Y0[i,j]
  :
  j00[i1,i1+N]:=Y0[i,j+1]+Y0[i,j]:j00[i1,i1-N]:=Y0[i,j-1]+Y0[i,j]
  :
od:
od:
for i from 1 to 1 do for j from 1 to M do
i1:=i+(j-1)*N:
j00[i1,i1]:=-1.0:j00[i1,i1+1]:=1.00:
od:od:
for i from N to N do for j from 1 to M do
i1:=i+(j-1)*N:
j00[i1,i1]:=-1.0:j00[i1,i1-1]:=1.00:
od:od:
for i from 1 to N do for j from 1 to 1 do
i1:=i+(j-1)*N:
j00[i1,i1]:=-1.0:j00[i1,i1+N]:=1.00:
od:od:
for i from 1 to N do for j from M to M do
i1:=i+(j-1)*N:
j00[i1,i1]:=-1.0:j00[i1,i1-N]:=1.00:
od:od:
NULL;
end proc:
```

First-order Upwind method is coded in the procedure UpW.

```
> UpW:=proc(Y00::Matrix(datatype=float[8]),Phi0::Vector(datatype=
float[8]),F0::Matrix(datatype=float[8]),dt::float,N::integer,
M::integer,v0::float)
local i::integer,j::integer,h::float[8],nx::float[8],vel::float
[8],vx::float[8],vy::float[8],phix::float[8],phiy::float[8],
phiave::float[8],jj::integer,phixb::float[8],phixf::float[8],
phixb2::float[8],phixf2::float[8],vxb::float[8],phiyb::float[8],
phiyf::float[8],vxf::float[8],phiyb2::float[8],phiyf2::float[8],
vyb::float[8],vyf::float[8],uf::float[8],ub::float[8],vf::float
[8],vb::float[8],tt::float[8],vv0::float[8],sd::::float[8],
sdf::float[8],sdb::float[8],sdx::float[8],sdy::float[8],
sdxb::float[8],s1x::float[8],sdxr::float[8],sdyb::float[8],
sdyf::float[8],s1y::float[8],vx1::float[8],vx2::float[8],
vy1::float[8],vy2::float[8],w1::float[8],w2::float[8],r1::float
[8],r2::float[8],alpha::float[8],beta::float[8];
option optimize, autocompile;
h:=1.0/(N-2):
vv0:=v0:
for i from 1 to N do Y00[i,1]:=Y00[i,2]:Y00[i,M]:=Y00[i,M-1]:od:
for j from 1 to M do Y00[1,j]:=Y00[2,j]:Y00[N,j]:=Y00[N-1,j]:od:
for i from 2 to N-1 do for j from 2 to M-1 do
vx:=0.0:vy:=0.0:phix:=0.0:phiy:=0.0:
vx1:=(Y00[i,j]-Y00[i-1,j])/h:vx2:=(Y00[i+1,j]-Y00[i,j])/h:
vy1:=(Y00[i,j]-Y00[i,j-1])/h:vy2:=(Y00[i,j+1]-Y00[i,j])/h:
if v0>=0 then
vx1:= max(vx1,0):vx2:=-min(vx2,0): else
vx1:=-min(vx1,0):vx2:= max(vx2,0): end:
if v0>=0 then
vy1:= max(vy1,0):vy2:=-min(vy2,0): else
vy1:=-min(vy1,0):vy2:= max(vy2,0): end:
nx:=sqrt(max(vx1,vx2)^2+max(vy1,vy2)^2):
F0[i,j]:=nx:
od:od:
end proc:
```

Second-order ENO2 method is coded in the procedure ENO2.

```
> ENO2:=proc(Y00::Matrix(datatype=float[8]),Phi0::Vector(datatype=
float[8]),F0::Matrix(datatype=float[8]),dt::float,N::integer,
M::integer,v0::float)
local i::integer,j::integer,h::float[8],nx::float[8],vel::float
[8],vx::float[8],vy::float[8],phix::float[8],phiy::float[8],
```

```

phiave::float[8],jj::integer,phixb::float[8],phixf::float[8],
phixb2::float[8],phixf2::float[8],vxb::float[8],phiyb::float[8],
phiyf::float[8],vxf::float[8],phiyb2::float[8],phiyf2::float[8],
vyb::float[8],vyf::float[8],uf::float[8],ub::float[8],vf::float
[8],vb::float[8],tt::float[8],vv0::float[8],sd::float[8],
sdf::float[8],sdb::float[8],sdx::float[8],sdy::float[8],
sdxb::float[8],s1x::float[8],sdxf::float[8],sdyb::float[8],
sdyf::float[8],sly::float[8],vx1::float[8],vx2::float[8],
vy1::float[8],vy2::float[8],alpha::float[8],beta::float[8];
option optimize, autocompile;
h:=1.0/(N-2):
for i from 1 to N do Y00[i,1]:=Y00[i,2]:Y00[i,M]:=Y00[i,M-1]:od:
for j from 1 to M do Y00[1,j]:=Y00[2,j]:Y00[N,j]:=Y00[N-1,j]:od:
for i from 2 to N-1 do for j from 2 to M-1 do
vx:=0.0:vy:=0.0:phix:=0.0:phiy:=0.0:
sdx:=(Y00[i+1,j]-2*Y00[i,j]+Y00[i-1,j])/h:sdy:=(Y00[i,j+1]-2*Y00
[i,j]+Y00[i,j-1])/h:
vxb:=0:vxf:=0:vyb:=0:vyf:=0:
if i = 2 then
sdxb:=(Y00[i,j]-2*Y00[i-1,j]+Y00[i-1,j])/h:else
sdxb:=(Y00[i,j]-2*Y00[i-1,j]+Y00[i-2,j])/h:end:
if sdx*sdxb>=0 then s1x:=1.0 else s1x:=0.0:end:
vx1:=(Y00[i,j]-Y00[i-1,j])/h+0.5*signum(sdx)*s1x*min(abs(sdx),abs
(sdxb)):
if i = N-1 then
sdxf:=(Y00[i+1,j]-2*Y00[i+1,j]+Y00[i,j])/h:else
sdxf:=(Y00[i+2,j]-2*Y00[i+1,j]+Y00[i,j])/h:end:
if sdx*sdxf>=0 then s1x:=1.0 else s1x:=0.0:end:
vx2:=(Y00[i+1,j]-Y00[i,j])/h-0.5*signum(sdx)*s1x*min(abs(sdx),abs
(sdxf)):
if j = 2 then
sdyb:=(Y00[i,j]-2*Y00[i,j-1]+Y00[i,j-1])/h:else
sdyb:=(Y00[i,j]-2*Y00[i,j-1]+Y00[i,j-2])/h:end:
if sdy*sdyb>=0 then sly:=1.0 else sly:=0.0:end:
vy1:=(Y00[i,j]-Y00[i,j-1])/h+0.5*signum(sdy)*sly*min(abs(sdy),abs
(sdyb)):
if j = M-1 then
sdyf:=(Y00[i,j+1]-2*Y00[i,j+1]+Y00[i,j])/h:else
sdyf:=(Y00[i,j+2]-2*Y00[i,j+1]+Y00[i,j])/h:end:
if sdy*sdyf>=0 then sly:=1.0 else sly:=0.0:end:
vy2:=(Y00[i,j+1]-Y00[i,j])/h+0.5*signum(sdy)*sly*min(abs(sdy),abs
(sdyf)):

```

```

if v0>=0 then
vx1:= max(vx1,0):vx2:=-min(vx2,0): else
vx1:=-min(vx1,0):vx2:= max(vx2,0): end:
if v0>=0 then
vy1:= max(vy1,0):vy2:=-min(vy2,0): else
vy1:=-min(vy1,0):vy2:= max(vy2,0): end:
nx:=sqrt(max(vx1,vx2)^2+max(vy1,vy2)^2):
F0[i,j]:=nx:
od:od:
end proc:

```

Third-order WENO3 method is coded in the procedure WENO3.

```

> WENO3:=proc(Y00::Matrix(datatype=float[8]),Phi0::Vector(datatype=
float[8]),F0::Matrix(datatype=float[8]),dt::float,N::integer,
M::integer,v0::float)
local i::integer,j::integer,h::float[8],nx::float[8],vel::float
[8],vx::float[8],vy::float[8],phix::float[8],phiy::float[8],
phiave::float[8],jj::integer,phixb::float[8],phixf::float[8],
phixb2::float[8],phixf2::float[8],vxb::float[8],phiyb::float[8],
phiyf::float[8],vxf::float[8],phiyb2::float[8],phiyf2::float[8],
vyb::float[8],vyf::float[8],uf::float[8],ub::float[8],vf::float
[8],vb::float[8],tt::float[8],vv0::float[8],sd::::float[8],
sdf::float[8],sdb::float[8],sdx::::float[8],sdy::::float[8],
sdxb::float[8],s1x::float[8],sdxr::float[8],sdyb::float[8],
sdyf::float[8],sly::float[8],vx1::float[8],vx2::float[8],
vy1::float[8],vy2::float[8],w1::float[8],w2::float[8],r1::float
[8],r2::float[8],alpha::float[8],beta::float[8],e1::float[8];
option optimize, autocompile;
e1:=1e-6:
nx:=0.0:
h:=1.0/(N-2):
vv0:=v0:
for i from 1 to N do Y00[i,1]:=Y00[i,2]:Y00[i,M]:=Y00[i,M-1]:od:
for j from 1 to M do Y00[1,j]:=Y00[2,j]:Y00[N,j]:=Y00[N-1,j]:od:
for i from 2 to N-1 do for j from 2 to M-1 do
vx:=0.0:vy:=0.0:phix:=0.0:phiy:=0.0:
phix:=(Y00[i+1,j]-Y00[i-1,j])/2/h:phiy:=(Y00[i,j+1]-Y00[i,j-1])
/2/h:
if i = 2 then
sdb:=Y00[i,j]-2*Y00[i-1,j]+Y00[i-1,j]: else
sdb:=Y00[i,j]-2*Y00[i-1,j]+Y00[i-2,j]:end:
sd:=Y00[i+1,j]-2*Y00[i,j]+Y00[i-1,j]:

```

```

if i = N-1 then
sdf:=Y00[i,j]-2*Y00[i+1,j]+Y00[i+1,j]: else
sdf:=Y00[i+2,j]-2*Y00[i+1,j]+Y00[i,j]:end:
r1:=(e1+sdb^2)/(e1+sd^2):w1:=1/(1+2*r1^2):
r2:=(e1+sdf^2)/(e1+sd^2):w2:=1/(1+2*r2^2):
vx1:=phix-0.5*w1/h*(sd-sdb):
vx2:=phix-0.5*w2/h*(sdf-sd):
if j = 2 then
sdb:=Y00[i,j]-2*Y00[i,j-1]+Y00[i,j-1]:else
sdb:=Y00[i,j]-2*Y00[i,j-1]+Y00[i,j-2]:end:
sd:=Y00[i,j+1]-2*Y00[i,j]+Y00[i,j-1]:
if j = M-1 then
sdf:=Y00[i,j]-2*Y00[i,j+1]+Y00[i,j+1]: else
sdf:=Y00[i,j+2]-2*Y00[i,j+1]+Y00[i,j]:end:
r1:=(e1+sdb^2)/(e1+sd^2):w1:=1/(1+2*r1^2):
r2:=(e1+sdf^2)/(e1+sd^2):w2:=1/(1+2*r2^2):
vy1:=phiy-0.5*w1/h*(sd-sdb):
vy2:=phiy-0.5*w2/h*(sdf-sd):
if v0>=0 then
vx1:=max(vx1,0):vx2:=-min(vx2,0): else
vx1:=-min(vx1,0):vx2:=max(vx2,0): end:
if v0>=0 then
vy1:=max(vy1,0):vy2:=-min(vy2,0): else
vy1:=-min(vy1,0):vy2:=max(vy2,0): end:
nx:=sqrt(max(vx1,vx2)^2+max(vy1,vy2)^2):
F0[i,j]:=nx:
od:od:
end proc:

```

```

> PhiAdd:=proc(N::integer,Phi0::Vector(datatype=float[8]),
db::Vector(datatype=float[8]))
local i::integer;
option optimize, autocompile;
for i from 1 to N do Phi0[i]:=Phi0[i]+db[i]:od:
end proc:
> phiaveAdd:=proc(Y00::Matrix(datatype=float[8]),N::integer,
M::integer,Ny::integer)
local i::integer,j::integer,phiave::float;
option optimize, autocompile;
phiave:=0.0:
for i from 2 to N-1 do for j from 2 to M-1 do phiave:=phiave+Y00
[i,j]:od:od:

```

```

phiave/(N-2)/(M-2);
end proc;

> EFAdd:=proc(Y00::Matrix(datatype=float[8]),Ymid::Matrix(datatype=
float[8]),Phi0::Vector(datatype=float[8]),F0::Matrix(datatype=
float[8]),dt::float,vel::float,ki0::float,N::integer,M::integer)
local i::integer,j::integer;
option optimize, autocompile;
for i from 2 to N-1 do for j from 2 to M-1 do Ymid[i,j]:=max
(1e-9,Y00[i,j]-dt*vel*ki0*F0[i,j]*Phi0[i+(j-1)*N]):od:od:
for i from 1 to N do Ymid[i,1]:=Ymid[i,2]:Ymid[i,M]:=Ymid[i,M-1]
:od:
for j from 1 to M do Ymid[1,j]:=Ymid[2,j]:Ymid[N,j]:=Ymid[N-1,j]
:od:
end proc;

> EFAdd2:=proc(Y00::Matrix(datatype=float[8]),Ymid::Matrix
(datatype=float[8]),Phi0::Vector(datatype=float[8]),F0::Matrix
(datatype=float[8]),dt::float,vel::float,ki0::float,N::integer,
M::integer)
local i::integer,j::integer;
option optimize, autocompile;
for i from 2 to N-1 do for j from 2 to M-1 do
Ymid[i,j]:=max(1e-9,Y00[i,j]*3/4.+Ymid[i,j]/4.-dt/4.*vel*ki0*F0
[i,j]*Phi0[i+(j-1)*N]):od:od:
for i from 1 to N do Ymid[i,1]:=Ymid[i,2]:Ymid[i,M]:=Ymid[i,M-1]
:od:
for j from 1 to M do Ymid[1,j]:=Ymid[2,j]:Ymid[N,j]:=Ymid[N-1,j]
:od:
end proc;

> EFAdd3:=proc(Y00::Matrix(datatype=float[8]),Ymid::Matrix
(datatype=float[8]),Phi0::Vector(datatype=float[8]),F0::Matrix
(datatype=float[8]),dt::float,vel::float,ki0::float,N::integer,
M::integer)
local i::integer,j::integer;
option optimize, autocompile;
for i from 2 to N-1 do for j from 2 to M-1 do
Y00[i,j]:=max(1e-9,Y00[i,j]*1/3.+Ymid[i,j]*2/3.-dt*2/3.*vel*ki0*
F0[i,j]*Phi0[i+(j-1)*N]):od:od:
for i from 1 to N do Y00[i,1]:=Y00[i,2]:Y00[i,M]:=Y00[i,M-1]:od:
for j from 1 to M do Y00[1,j]:=Y00[2,j]:Y00[N,j]:=Y00[N-1,j]:od:
end proc;

> jtot:=(N-2)*4*2+4+(N-2)*(M-2)*5:
> TT[0]:=0:tt:=0:

```

```

> #j00:=Matrix(1..ntot,1..ntot,datatype=float[8],storage=sparse):
F0   :=Matrix(1..N,1..M,datatype=float[8]):
Phi0:=Vector(1..ntot,datatype=float[8]):
ff:=copy(Phi0):
> Aii:=Vector(jtot,datatype=integer[4]):Api:=Vector(ntot+1,
datatype=integer[4]):
jss:=Vector(jtot,datatype=float[8]):
> ApAiproc(Api,Aii,N,M);
                                         50804
> Api:=convert(Api,Vector,datatype=integer[8]):Aii:=convert(Aii,
Vector,datatype=integer[8]):
> evalf(Eqs11(Y00,Phi0,delta,ki0,ff,N,M)):
> jsproc(Y00,Phi0,delta,ki0,jss,N,M);
                                         -1.
> #Jac1(Y00,Phi0,delta,ki0,j00,N,M):
> #with(LinearAlgebra):
> umfsym:=define_external("umfpack_dl_symbolic",
'm'::(integer[8]),
'n'::(integer[8]),
'Ap'::ARRAY(integer[8]),
'Ai'::ARRAY(integer[8]),
'Ax'::ARRAY(float[8]),
'Sym'::REF(integer[8]),
'Control'::ARRAY(float[8]),
'Info'::ARRAY(float[8]),
LIB="umfpack_dlong"):

> umfnum:=define_external("umfpack_dl_numeric",
'Ap'::ARRAY(integer[8]),
'Ai'::ARRAY(integer[8]),
'Ax'::ARRAY(float[8]),
'Sym'::(integer[8]),
'Num'::REF(integer[8]),
'Control'::ARRAY(float[8]),
'Info'::ARRAY(float[8]),
LIB="umfpack_dlong"):

> umfdefaults:=define_external("umfpack_dl_defaults",
'Control'::ARRAY(float[8]),
LIB="umfpack_dlong"):

> Control:=Vector[row](1..20,datatype=float[8]):
> umfdefaults(Control);

```

```

> #umfdefaults;
> seq(Control[i], i=1..20);
1., 0.200000000000000, 0.200000000000000, 0.100000000000000, 32., 0., 0.700000000000000, (5)
  2., 0., 0., 0., 0.010000000000000, 0., 10., 0.001000000000000, 1., 0.500000000000000,
  0., 1.

> #Control[5]:=64.;

> #seq(Control[i], i=1..4);

> Info:=Vector[row](1..91, datatype=float[8]): 

> extlib:=ExternalCalling:-ExternalLibraryName("linalg", 'HWFfloat'):

umfSolve:=ExternalCalling:-DefineExternal(':-

hw_SpUMFPACK_MatVecSolve', extlib):

> umfsym(ntot, ntot, Api, Aii, jss, 'Sym', Control, Info):

> umfnum(Api, Aii, jss, Sym, 'Num', Control, Info):

> db:=Copy(Phi0):

> db:=umfSolve(ntot, jtot, Num, -ff):

> #extfun:=ExternalCalling:-DefineExternal(':-

hw_SpUMFPACK_FreeNumeric' ,

#extlib):

> PhiAdd(ntot, Phi0, db):

> v[0]:=(Phi0[ntot-2*N+N/2]/2+Phi0[ntot-2*N+N/2+1]/2)+h/2*delta;
V_0 := 0.315812378860094 (6)

> #extfun(Num);

> Phiave[0]:=phiaveAdd(Y00, N, M, MM);

Phiave_0 := 0.929280193991299241 (7)

```

```

> vv0:=max(abs(Phi0[ntot-2*N+1]), abs(Phi0[ntot-2*N+N/2]), abs(Phi0
  [ntot-2*N+N/2+1]), abs(Phi0[ntot-N])):

> dt:=min(h/vv0/vel/ki0, tf-tt);
dt := 0.0312027709947108 (8)

```

The three stages of SSR-RK3 are stored in EFAdd, EFAdd2 and EFAdd3

```

> YdatStore:=proc(Y00::Matrix(datatype=float[8]), Ydat::Array
  (datatype=float[8]), N::integer, M::integer, jj::integer)
local i::integer, j::integer;
option optimize, autocompile;
for i from 2 to N-1 do for j from 2 to M-1 do
Ydat[jj,i,j]:=Y00[i,j]:od:od:
end proc;

> Nt:=round(tf/dt)+50;

```

$$Nt := 114 \quad (9)$$

```

> Ymid:=copy(Y00):
> Ydat:=Array(1..Nt+1,1..N,1..M,datatype=float[8]):
> YdatStore(Y00,Ydat,N,M,1):

```

Different upwind schemes can be called by assign WENO3, UpW or ENO3 scheme.

```
> HF:=eval(WENO3):
```

A while loop is written from t=0 to t= tf.

```

> ii:=0:
  while tt<tf do
    HF(Y00,Phi0,F0,evalf(dt),N,M,delta):
    EFAdd(Y00,Ymid,Phi0,F0,dt,vel,ki0,N,M);
    Eqs11(Ymid,Phi0,delta,ki0,ff,N,M);
    jsproc(Ymid,Phi0,delta,ki0,jss,N,M);
    umfnum(Api,Aii,jss,Sym,'Num',Control,Info);
    db:=umfSolve(ntot,jtot,Num,-ff):#extfun(Num);Num:=-1:
    PhiAdd(ntot,Phi0,db):
    HF(Ymid,Phi0,F0,evalf(dt),N,M,delta):
    EFAdd2(Y00,Ymid,Phi0,F0,dt,vel,ki0,N,M);
    Eqs11(Ymid,Phi0,delta,ki0,ff,N,M);
    jsproc(Ymid,Phi0,delta,ki0,jss,N,M);
    umfnum(Api,Aii,jss,Sym,'Num',Control,Info);
    db:=umfSolve(ntot,jtot,Num,-ff):#extfun(Num);Num:=-1:
    PhiAdd(ntot,Phi0,db):
    HF(Ymid,Phi0,F0,evalf(dt),N,M,delta):
    EFAdd3(Y00,Ymid,Phi0,F0,dt,vel,ki0,N,M);
    Eqs11(Y00,Phi0,delta,ki0,ff,N,M);
    jsproc(Y00,Phi0,delta,ki0,jss,N,M);
    umfnum(Api,Aii,jss,Sym,'Num',Control,Info);
    db:=umfSolve(ntot,jtot,Num,-ff):#extfun(Num);Num:=-1:
    PhiAdd(ntot,Phi0,db):
    ii:=ii+1:
    V[ii]:=(Phi0[ntot-2*N+N/2]/2+Phi0[ntot-2*N+N/2+1]/2)+h/2*delta;
    #print(ii,V[ii]);
    TT[ii]:=TT[ii-1]+dt:tt:=tt+dt:
    vv0:=max(abs(Phi0[ntot-2*N+1]),abs(Phi0[ntot-2*N+N/2]),abs(Phi0[ntot-2*N+N/2+1]),abs(Phi0[ntot-N])):
    dt:=min(h/vv0/vel/ki0,tf-tt);
    #YdatStore(Y00,Ydat,N,M,ii+1);
    Phiave[ii]:=phiaveAdd(Y00,N,M,MM);
    gc();

```

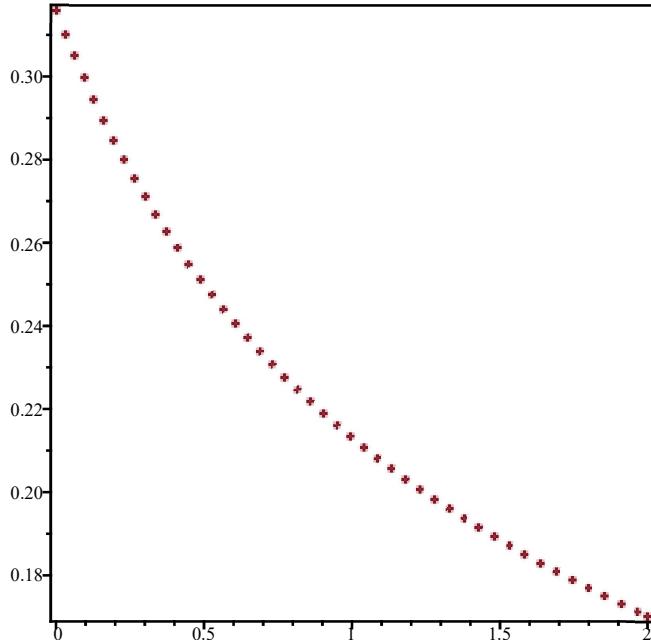
```
| end:
```

```
|> nt:=ii;
          nt := 46
```

 (10)

Voltage time curves are plotted below. Voltage is measured at X = 0.5, Y = 1.

```
|> plot([[seq([TT[ii],V[ii]],ii=0..nt)]],style=point,axes=boxed);
```



```
|> v[0]:v[2];
          0.315812378860094
          0.304922457763894
```

 (11)

Voltage at the end of plating, cpu time can be documented as

```
|> [NN,time[real]()-t12,time()-t11,V[nt]];
          [100, 8.714, 6.329, 0.170214283284059]
```

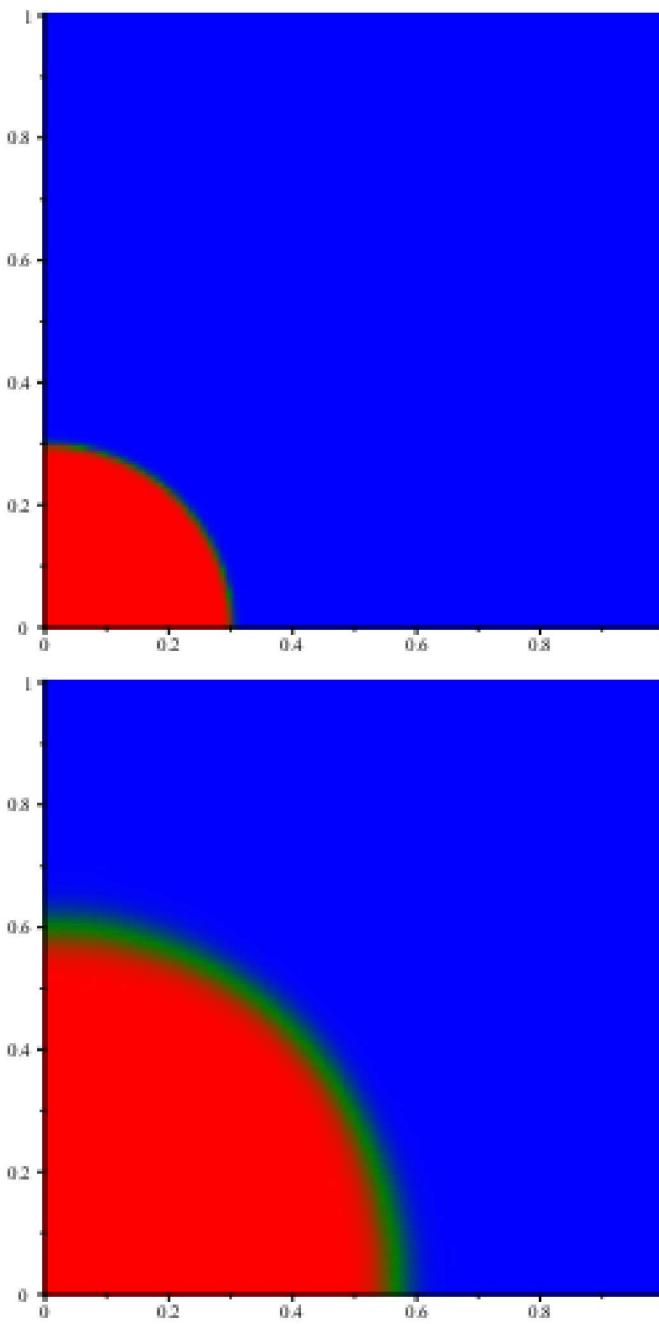
 (12)

```
|> p1:=plots:-surfdata(Y00,-h/2..NN*h+h/2,-h/2..MM*h+h/2,dimension=
           2,style=surface,colorscheme = ["Red", "Green", "Blue"]):
|> tf:=TT[nt];
          tf := 2.
```

 (13)

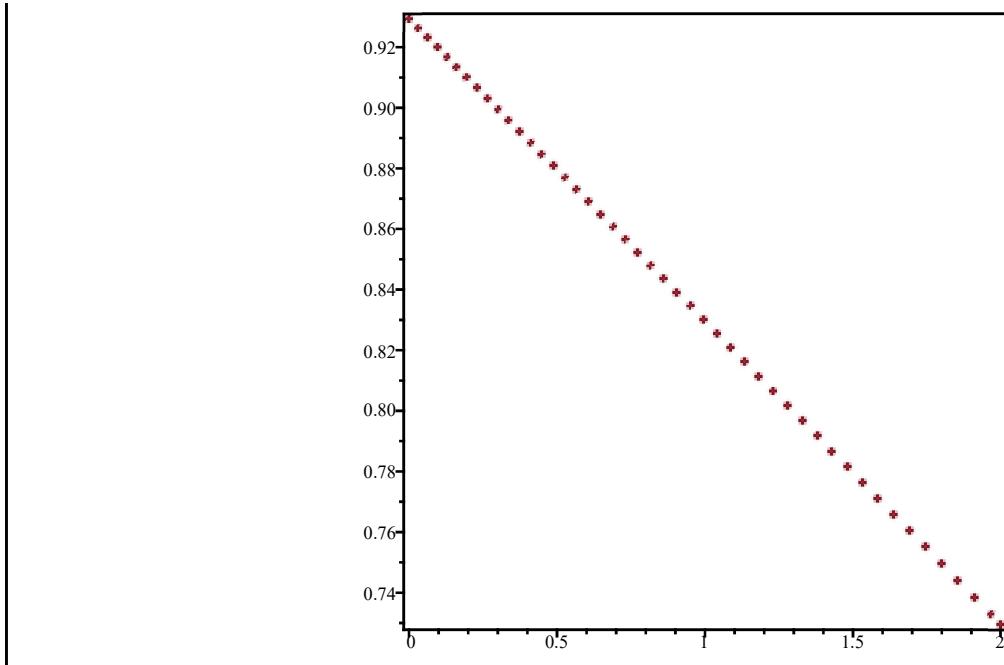
Contour plots at t= 0 and t = 2.0 (at the end of plating are given below)

```
|> plots:-display({p0});plots:-display({p1});
```



The liquid phase content as a function of time is plotted below

```
> plot([[seq([TT[ii],Phiave[ii]],ii=0..nt)]],style=point,axes=boxed);
```



```
[> save Y00,"Y0data.m";
> save tf,"tdata.m";
```