Numerical Methods in Engineering – Part I

Dr Dorival Pedroso

March 14, 2017

Introduction

Introduction

- Course goal: to **program** the FDM and FEM using Python
- Python is a *scripting language* $\Rightarrow$ it is **interpreted** (not compiled), does not require variables declaration, and has a *high-level* regarding machine-programmer "interaction"
- Why Python ?
- Answer: **easy and beautiful!**
- Even better: it is free software with open source and has high extensibility (huge extensive library)
- Moreover: MIT teaches Python, NASA uses Python, and Google uses Python! ...
- Finally: Many programs use Python for internal scripting, e.g. MechSys, Blender, ParaView, GIMP, and ABAQUS! ...

Bibliography for Python

- The Python Tutorial: `http://docs.python.org/tutorial/index.html`
- NumPy and SciPy: `http://www.scipy.org/Getting_Started`
- NumPy and SciPy: `http://docs.scipy.org/doc/scipy/reference/tutorial`
- MatPlotLib: `http://matplotlib.org/api/pyplot_summary.html`

Fundamentals

*nix and Linux

- Unix: Developed at Bell Labs by Ken Thompson, Dennis Ritchie and co-workers https://en.wikipedia.org/wiki/Unix
- Linux: Linus Torvalds (see also the idea by Richard Stallman) https://en.wikipedia.org/wiki/Linux
- Linux distros: Red Hat (Fedora), Debian, Ubuntu https://en.wikipedia.org/wiki/Linux_distribution
- Apple's kernel: Darwin (Berkley Software Distribution – BSD/*nix)
  https://en.wikipedia.org/wiki/Darwin_(operating_system
  https://en.wikipedia.org/wiki/Berkeley_Software_Distri
- Other 'unixes': Apple iOS (Darwin-based), Apple Mac OS X (Darwin-based), Google Android (Linux-based)
  https://en.wikipedia.org/wiki/IOS
  https://en.wikipedia.org/wiki/OS_X
  https://en.wikipedia.org/wiki/Android_(operating_syste

Ubuntu Linux

- Ubuntu is one of the nicest Linux distro around.
- You can freely download it from `http://www.ubuntu.com/desktop`
- Or better, from the Australian's Academic and Research Network (AARNet): `https://mirror.aarnet.edu.au/`

Computer programming languages

- C programming language (developed by Dennis Ritchie; receiver of the Turing Award, the Hamming Medal and the National Medal of Technology by the US president) `https://en.wikipedia.org/wiki/C_(programming_language)`
- Java (developed by Sun Microsystems; now Oracle) `https://en.wikipedia.org/wiki/Java_(programming_langua`
- C and Java are the two most popular languages around
- Python (developed by Guido van Rossum/worked at Google)
- Go language (developed by Google engineers, including Robert Griesemer, Rob Pike, Ken Thompson, and co-workers) `https://golang.org/`
- Ken Thompson is the same one that created Unix. He was also awarded the Turing Award, National Academy of Engineering, National Medal of Technology by the US president, and others! `https://en.wikipedia.org/wiki/Ken_Thompson`

Other interesting/important computer scientists

- ☐ First mechanical computer created by Charles Babbage
  https://en.wikipedia.org/wiki/Charles_Babbage
- ☐ The first computer programmer (?): Ada Lovelace
  https://en.wikipedia.org/wiki/Ada_Lovelace
- ☐ Computer pioneer: Alan Turing
  https://en.wikipedia.org/wiki/Alan_Turing
- ☐ Great computer scientist: Donald Knuth (also a receiver of the Turing Award)
  https://en.wikipedia.org/wiki/Donald_Knuth
- ☐ The concept of 'bug', 'mistake' or 'faults' in computer Programming dates back to Ada Lovelace's notes
  https://en.wikipedia.org/wiki/Software_bug

History

1920: Torres Quevedo's electromechanical arithmometer

Numbers

- *Integers* ($\mathbb{Z}$) are the whole numbers [1]

$$\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$$

- *Natural* numbers ($\mathbb{N}$) are the non-negative integers (or sometimes the positive integers)
- A *real number* ($\mathbb{R}$) is a quantity x that has a *decimal expansion* [1]

$$x = n + 0.d_1 d_2 d_3 \ldots$$

where n is an integer and each d_i is a digit between 0 and 9.

- A *floating point number* is the approximated version of a *real* number using the following formula

$$s \times b^e$$

where $s \in \mathbb{Z}$ is the *significand*, $b \in \mathbb{N}$ is the *base*, and $e \in \mathbb{Z}$ is the *exponent*. https://en.wikipedia.org/wiki/Floating_point

References

[1] Knuth D (1997) The Art of Computer Programming–Volume 1: Fundamental Algorithms. 3rd Edition. Addison Wesley. 652 p.

Quick, Informal and Brief Introduction to Programming in Python

Code structure and scopes

- In Python, each scope is defined with *white spaces to the left* ⇒ **indentation**, followed by the colon **:** symbol
- There is no need for "begin" or "end" keywords or braces when defining scopes
- In the following figure, gray boxes are in the **global** scope
- Yellow boxes are in a **local** scope
- And light-blue boxes are in a **sub-local** scope (and so on)
- Commands that define scopes are: **class**, **def**, **if**, **else**, **elif**, **while**, and **for**

```
var = 1.0
var = 2.0
var = 3.0
for f in res:
    var = 4.0
    var = 5.0
    while True:
        var = 1
        var = 2
var = 6.0
var = 7.0
```


Declaration of variables and features

- Python is **casesensitive** ≠ **CaseSensitive**
- In Python, variables are declared simply with "**=**"
- The type/content of each variable is determined automatically (by the right-hand side of the expression)
- "**1**" is integer. "**1.0**" is float
- More than one variable can be declared in a single line. Ex.: x, y = 1, 2
- "**print**()" is a function for printing to the **Console**

```
aint    = 1
afloat  = 1.0
astring = 'HELLO'
alist   = [1, 2, 3]
x, y    = 1.0, 2.0
print 1/2
print astring
print x
print y
print 'x =',x,', y =',y
```

```
0
HELLO
1.0
2.0
x = 1.0 , y = 2.0
```

Python Lists. Symbol: `[]`

- ☐ Lists are **arrays** of things (numbers, characters, other objects, . . .)
- ☐ They are useful for grouping data
- ☐ Lists can have mixed "things" (objects). Example: `mylist=[1,'a', ['innerlist']]`
- ☐ To access data, use `[i]`, with `i` starting at 0

```python
#             0    1     2
velocity = [0.0, 0.0, -1.0]
tags     = ['first', 'second']
mix      = [1, 1.0, 'a']
print velocity
print tags
print mix
print velocity[2]
```

```
[0.0, 0.0, -1.0]
['first', 'second']
[1, 1.0, 'a']
-1.0
```


Python Lists. Symbol: `[]` (cont.)

- ☐ Since lists can have objects, we can create lists of lists "**[[ ]]**"
- ☐ **Slice** notation ⇒ "i:j", where "i" is the first index and "j-1" the last index
- ☐ [-1] ⇒ the last character
- ☐ [-2] ⇒ the last-but one character

```python
nums = [8,9,10,11, [100,200]]
print nums[1:3]
print nums[4]
print nums[-1]
print nums[-2]
```

```
[9, 10]
[100, 200]
[100, 200]
11
```

- Strings are lists
- [:-2] ⇒ everything except the last two characters
- [-2:] ⇒ the last two characters
- **Slice** notation ⇒ x = x[:i] + x[i:]
- "**len**()" is a function that calculates the "length" (content, size, ...) of an object

```
word = 'PYTHON'
list = ['P', 'Y', 'T', 'H', 'O', 'N']
print word[:-2]
print word[-2:]
print word[:-2] + word[-2:]
print len(word)
print len(list)
```

```
PYTH
ON
PYTHON
6
6
```


- "**range**(n)" is a function that generates a List with values from 0 to n-1
- "**range**(i,j)" generates a List with values from i to j-1

```
res = range(10)
vals = range(3,10)
print res
print vals
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[3, 4, 5, 6, 7, 8, 9]
```

Decision structures (`and`, `or`, `not`, `if`, `else`, `elif`)

- Keywords: **and**, **or**, **not**, **if**, **else**, **elif**
- Put ":" at the end of a line with **if**, **else**, **elif**
- Syntax: **if** True: ⇒ do something

Operator	Meaning
$>$	Greater than
$<$	Less than
$>=$	Greater than or equal to
$<=$	Less than or equal to
$==$	Equal to
$!=$	Not equal to
and	"and" keyword
or	"or" keyword
not	"not" keyword
True	Boolean "true"
False	Boolean "false"

```
if True: print 'yes'
else:    print 'no'

x = 0.5
if x>0.0 and x<1.0:
    print '(0.0<x<1.0)'
elif x<0.0:
    print '(x<0.0)'
elif x>1.0:
    print '(x>1.0)'
else:
    print '(x==0.0 or x==1.0)'

print 'x = ', x
```

```
yes
(0.0<x<1.0)
x =  0.5
```


Decision structures: in-line conditional expression

- When the decision structure is short, we can use an **in-line conditional expression**
- Syntax: `var = 0.0` **if** `True` **else** `1.0`
- That means: *var gets 0.0 if True, otherwise it gets 1.0*

```
flag = True
var  = 0.0 if flag else 1.0
print var
```

```
0.0
```

Repetition structures (`in`, `for`, `while`)

- Keywords: **in**, **for**, **while**
- Put ":" at the end of a line with **for** or **while**
- Syntax: **for** value **in** List**:**
- Syntax: **while** True**:**

Operator	Meaning
$+=$	Increment
$-=$	Decrement

```
for item in ['first', 'second']:
    print item

for i in range(3):
    print i

x = 0.0
while x < 0.2:
    x += 0.1
    print x
```

```
first
second
0
1
2
0.1
0.2
```


List comprehensions

- List comprehension is a construct for creating a list based on existing lists
- It follows the form of the mathematical **set-builder notation**
- Ex.: $S = \{2x | x \in \mathbb{N}, x^2 > 3\}$
- Which reads: *S is the set of all 2 times x where x is an item in the set of natural numbers ($\mathbb{N}$), for which x squared is greater than 3*

```
nums = [2*x for x in range(6)]
print nums

myset = [2*x for x in range(6) if x**2>3]
print myset
```

```
[0, 2, 4, 6, 8, 10]
[4, 6, 8, 10]
```

Lists – `enumerate()`

- ☐ "**enumerate**()" is a function that generates **pairs** of indices and items

```
birds = ['Magpie','Cocatoo']
for index, bird in enumerate(birds):
    print index, bird

cars = ['BMW','Audi']
for i, car in enumerate(cars):
    print i, car
```

```
0 Magpie
1 Cocatoo
0 BMW
1 Audi
```


Declaration of functions (`def, return`)

- ☐ Define functions using **def**
- ☐ Put ":" at the end of a line with **def**
- ☐ Return variables using **return**

```
# function declaration
def fun(x,y):
    return x + y

# function declaration
def scale(x,y):
    return x**2.0, y-x

# code in global scope
res  = fun   (2.0, 3.0)
m, n = scale (2.0, 3.0)
print res
print m, n
```

```
5.0
4.0 1.0
```

Function arguments/parameters

- Some arguments to functions may be optional and have default values
- This is done in the function declaration where optional/default parameters are defined using '='
- when calling a function, the non-default arguments must be all given
- If one of the latest optional parameters is given, all intermediate optional parameters must be given as well

```python
def func(xa, xb, xc, optA='one', optB=2.0, optC=[]):
    """
    func performs a computation on x1,x2,x3
    =======================================
    this function takes 6 arguments:
    x1 -- first 'x' component
    x2 -- second 'x' component
    x3 -- third 'x' component
    opt1 -- first optional arg; a string
    opt2 -- second optional arg; a real number
    opt3 -- third optional arg; a list
    """
    pass

res1 = func(1, 2, 3)
res2 = func(1, 2, 3, 'two')
#res3 = func(1, 2, 3, 3.0) ### <<<< INCORRECT
res3 = func(1, 2, 3, 'three', 3.0)
res4 = func(1, 2, 3, 'four', 4.0, ['a','b','c'])
res4 = func(1, 2, 3, optC=['a', 'b', 'c'])
```


Lambda (anonymous) functions

- Lambda functions are those that are not bound to a name
- They are useful in some scenarios, for instance:
  - To define a function to "use just once" (e.g. in `sorted`)
  - When you want to pass a function as an input argument to another function

```python
a = sorted([1,2,3,4,5])
b = sorted([1,2,3,4,5], key=lambda x: abs(5-x))
print 'a =', a
print 'b =', b

def myFunc(x, anotherFunc):
    return anotherFunc(x) + 1

c = myFunc(3, lambda v: v**2.0)
print 'c =', c
```

```
a = [1, 2, 3, 4, 5]
b = [5, 4, 3, 2, 1]
c = 10.0
```

Dictionaries. Symbol {}

Introduction

Fundamentals

Python
1: Scopes
2: Variables
3: Lists 1
4: Lists 2
5: Lists 3
6: Lists 4
7: Decision 1
8: Decision 2
9: Repetition
10: List comp
11: List enum
12: Functions
13: Func. Args
14: Func. Lambda
▷ 15: Dictionaries
16: Tuples
17: OOP
18: Classes

Eclipse IDE

Plotting

- Dictionaries are maps between **keys** and **values** separated by "**:**"
- Dictionaries are created using "{" and "}"
- "**iteritems()**" is a **method** of dictionaries to iterate through the **pair** of keys and values

```
adrbook = { 'Dorival' :33653745,
            'CivilFAX':33654599 }

print 'Dorival  :', adrbook['Dorival']
print 'CivilFAX :', adrbook['CivilFAX']

for key, val in adrbook.iteritems():
    print key, val
```

```
Dorival  : 33653745
CivilFAX : 33654599
Dorival 33653745
CivilFAX 33654599
```


Tuples. Symbol ()

Introduction

Fundamentals

Python
1: Scopes
2: Variables
3: Lists 1
4: Lists 2
5: Lists 3
6: Lists 4
7: Decision 1
8: Decision 2
9: Repetition
10: List comp
11: List enum
12: Functions
13: Func. Args
14: Func. Lambda
15: Dictionaries
▷ 16: Tuples
17: OOP
18: Classes

Eclipse IDE

Plotting

- Tuples are like lists but **immutable**
- They are useful when we need to group a pair, a triple, a quadruple, a 5-tuple, a n-tuple, ...
- Since tuples are immutable, they can be used as keys in dictionaries!

```
x0     = 0.0
y0     = 1.0
point  = (x0,y0)
points = {point:'A', (1.0,1.0):'B'}
print point
print points[point]
# point[0] = 1.0
# TypeError: 'tuple' object does not support item assignment
```

```
(0.0, 1.0)
A
```

Object Oriented Programming (OOP)

- OOP is a programming paradigm based on **classes** and **objects**
- OOP has advantages and disadvantages. Among its advantages, the concept of *encapsulation* and *modularity* are the best

```
title = '   numerical methods   '

print title.strip()
print title.upper()
```

```
numerical methods
   NUMERICAL METHODS
```

- Objects are elements (things) in your program. Ex: integers, floats, strings, lists, dictionaries, functions, . . .
- "*Calling elements of a program objects is a metaphor – a useful way of thinking about them*" (*Intro to OOP with Python*)
- To access **data** or **methods** of an object, use the dot "**.**" notation (which means "please"). Ex.: `myobject.sayhello()`

Declaration of classes (`class`)

- Class is a **recipe** or **formula** for making/creating new objects
- Put ":" at the end of a line with **class**
- In Python, you have to add the **self** keyword to **all** methods
- The **self** keyword is used to access the data/methods of an *already allocated* object (this object)
- The number of input arguments in methods will always be added to 1 because of *self*

```
# class declaration
class Dog():
    def __init__(self,name):
        self.name = name
    def bark(self):
        print self.name, 'barks'

# code in global scope
mydog = Dog('Rex')
mydog.bark()
```

```
Rex barks
```

Eclipse – Integrated Development Environment (IDE)

Remarks

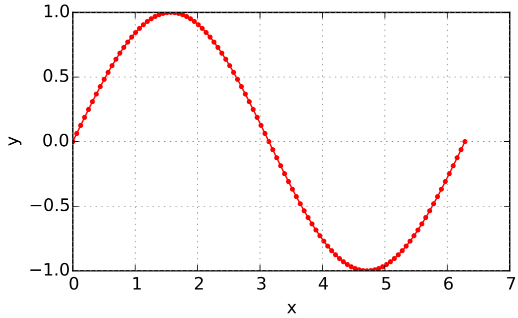
- Create new projects using PyDev; see installation instructions on accompanying file "*InstallingEclipseAndPylab.pdf*"
- Remember to add **.py** to your new filenames
- Create your own *toolbox.py* with your routines (or many other auxiliary files) ⇒ *your very own computing library*
  https://en.wikipedia.org/wiki/Library_(computing)
- Keep your library files clean
- Use the following code to keep the testing commands in your library from being imported

```
# my function
def MyFunction(x, y): return x + y

# just testing
if __name__ == "__main__":
    result = MyFunction(2, 2)
    if not result == 4: print 'test failed'
```

Plotting with MatPlotLib/PyPlot

Fundamentals

- Using NumPy and PyPlot
- NumPy helps with handling vectors and matrices, including generating numbers among many other features
- PyPlot helps with plotting 2D and 3D graphs




```python
import numpy as np
import matplotlib.pyplot as plt

def Gll(xname, yname):
    """
    Gll adds grid, legend and labels
    ================================
    Notes:
      zorder -- keep grid below curves/points
      best   -- best place for legend
    """
    plt.grid(color='grey', zorder=-100)
    plt.xlabel(xname)
    plt.ylabel(yname)
    plt.legend(loc='best', fontsize=8)

x = np.linspace(0.0, 2.0*np.pi, 101)
y = np.sin(x)
plt.plot(x, y, 'r.-')
Gll('x', 'y')
plt.show()
```

Numerical Methods in Engineering – Part II

Dr Dorival Pedroso

March 13, 2017

Fundamentals

Newton-Raphson's method

Root solver: Newton-Raphson's method

Suppose we want to find the roots x of:

$$r(x) = 0 \quad (\text{ex: } a + b\,x + c\,x^2 = 0)$$

We start with an initial guess x_k *close enough* to the unknown solution and by Taylor's series expansion:

$$r(x_k + \delta x) = r(x_k) + \left.\frac{\mathrm{d}r}{\mathrm{d}x}\right|_{x_k} \delta x + \left.\frac{\mathrm{d}^2 r}{\mathrm{d}x^2}\right|_{x_k} \frac{(\delta\, x)^2}{2} + O((\delta x)^3)$$

We require that $r(x_k + \delta\, x) = 0$. Thus, in addition to truncating higher order terms, we get:

$$r(x_k) + \left.\frac{\mathrm{d}r}{\mathrm{d}x}\right|_{x_k} \delta x = 0 \qquad \Rightarrow \qquad \delta x = -\left[\left.\frac{\mathrm{d}r}{\mathrm{d}x}\right|_{x_k}\right]^{-1} r(x_k)$$

Then, a better approximation of x_k is:

$$x_{k+1} = x_k + \delta x$$

Root solver: Newton-Raphson's method (cont.)

Example: $r(x) = x^2 - 4$

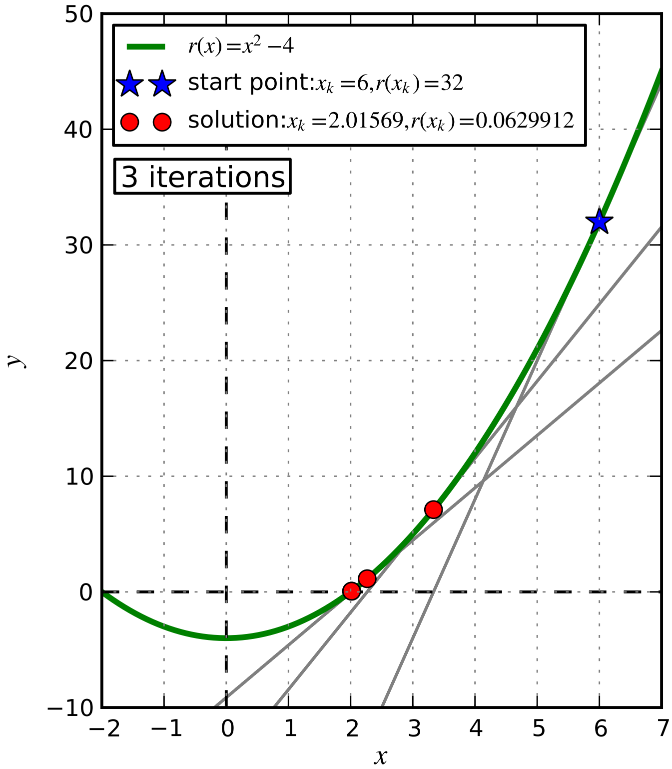
With: $J(x) = \left.\frac{\mathrm{d}r}{\mathrm{d}x}\right|_x = 2\,x$

Python implementation:

```
def r(x): return x**2.0-4.0
def J(x): return 2.0*x
xk = 6.0
for it in range(3):
    dx = - r(xk) / J(xk)
    xk += dx
print 'xk    =', xk
print 'r(xk) =', r(xk)
```

Results:

```
xk    = 2.01568627451
r(xk) = 0.0629911572472
```



Classification of linear partial differential equations (PDEs)

Quadratic equation and conics

Fundamentals
Newton-Raphson
Newton-Raphson 1
Newton-Raphson 2
PDEs class.
▷ Quadrics
PDEs and Analogy
PDEs: examples
1D FDM
1D Stability
1D Poisson
1D Diffusion
1D Wave
2D Poisson
FDM Conclusions

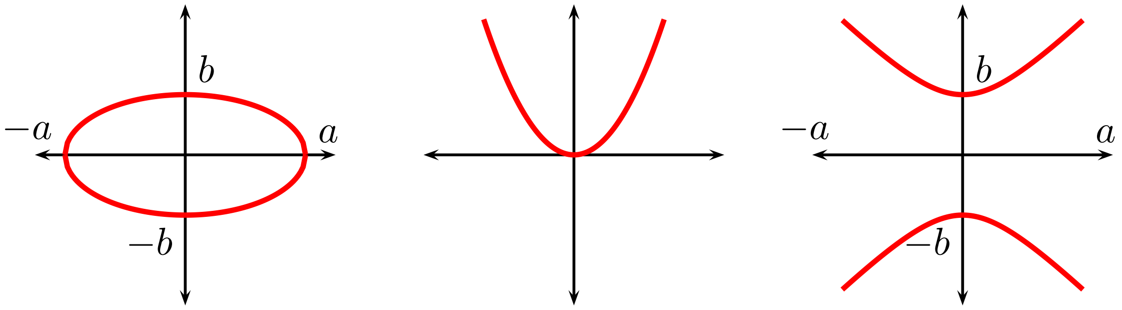
In Cartesian coordinates, the generalised quadratic equation is:

$$A\,x^2 + B\,x\,y + C\,y^2 + D\,x + E\,y + F = 0$$

For $\Delta = B^2 - 4\,A\,C$:

- ☐ If $\Delta < 0$, the equation represents an ellipse
- ☐ If $\Delta = 0$, the equation represents a parabola
- ☐ If $\Delta > 0$, the equation represents a hyperbola

Examples:



Classification of linear PDEs: Analogy with quadrics

Fundamentals
Newton-Raphson
Newton-Raphson 1
Newton-Raphson 2
PDEs class.
Quadrics
▷ PDEs and Analogy
PDEs: examples
1D FDM
1D Stability
1D Poisson
1D Diffusion
1D Wave
2D Poisson
FDM Conclusions

Family of second order (linear) PDEs:

$$A\,\frac{\partial^2\phi}{\partial x^2} + B\,\frac{\partial^2\phi}{\partial x \partial y} + C\,\frac{\partial^2\phi}{\partial y^2} + D\,\frac{\partial\phi}{\partial x} + E\,\frac{\partial\phi}{\partial y} + F = 0$$

Classification ($\Delta = B^2 - 4\,A\,C$):

- ☐ $\Delta < 0$: Elliptic $\Rightarrow$ Boundary Value Problems (BVPs)
- ☐ $\Delta = 0$: Parabolic $\Rightarrow$ Initial Boundary Value Prob. (IBVPs)
- ☐ $\Delta > 0$: Hyperbolic $\Rightarrow$ Initial Boundary Value Prob. (IBVPs)

Examples, with $s = s(x, y)$:

Poisson's equation:

$$\boxed{\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = -s}$$

(elliptic)

Diffusion equation:

$$\boxed{\frac{\partial\phi}{\partial t} - \frac{\partial^2\phi}{\partial x^2} - \frac{\partial^2\phi}{\partial y^2} = s}$$

(parabolic)

Wave equation:

$$\boxed{\frac{\partial^2\phi}{\partial t^2} - \frac{\partial^2\phi}{\partial x^2} - \frac{\partial^2\phi}{\partial y^2} = s}$$

(hyperbolic)

Some linear PDEs: examples

With $\phi_t = \frac{\partial \phi}{\partial t}$, $\phi_{xx} = \frac{\partial^2 \phi}{\partial x^2}$ and $\phi_{tt} = \frac{\partial^2 \phi}{\partial t^2}$

Poisson's and diffusion/heat:

$$-k_x\,\phi_{xx} = s \qquad \phi = \phi(x)$$

$$\phi_t - k_x\,\phi_{xx} = s \qquad \phi = \phi(t, x)$$

Geometry: a rod or wire fully insulated along its length or an impermeable cylinder filled with a porous medium (sand with water)

ϕ is the temperature T or hydraulic head H

k_x represents thermal diffusivity, conductivity, etc.

$s(x)$ is a source term, for instance a heat source, heat generation, water production, sink, recharging, pumping

Wave equation:

$$\rho\,\phi_{tt} - \sigma\,\phi_{xx} = s \qquad \phi = \phi(t, x)$$

Geometry: a string fixed to both ends with and applied transversal push

For instance, ϕ is the lateral displacement of a stretched string

$s(x)$ represents a transversal distributed force

ρ is mass per unit length and f the longitudinal tension. These define $c = \sqrt{\frac{\sigma}{\rho}}$ as the *wave speed*

Finite differences in one dimension (1D)

Let's approximate the derivative of $y(x)$ at node **n** using finite differences:

Forward difference:

$$\left.\frac{\mathrm{d}y}{\mathrm{d}x}\right|_{x_n} \approx \frac{y_r - y_n}{\Delta x}$$

Backward difference:

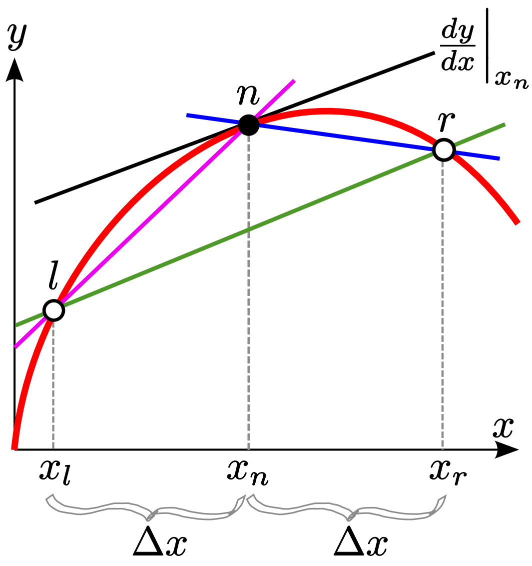
$$\left.\frac{\mathrm{d}y}{\mathrm{d}x}\right|_{x_n} \approx \frac{y_n - y_l}{\Delta x}$$

Central difference:

$$\left.\frac{\mathrm{d}y}{\mathrm{d}x}\right|_{x_n} \approx \frac{y_r - y_l}{2\Delta x}$$

l $\Rightarrow$ left of **n**

r $\Rightarrow$ right of **n**



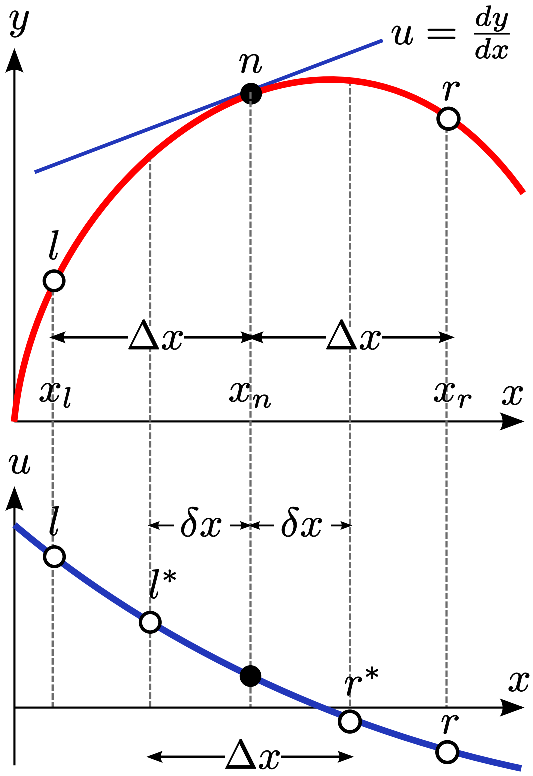
The same can be done for higher order derivatives. For example, by introducing an auxiliary variable u such that $u = \frac{\mathrm{d}y}{\mathrm{d}x}$:

$$\frac{\mathrm{d}u}{\mathrm{d}x} = \frac{\mathrm{d}^2 y}{\mathrm{d}x^2} \approx \frac{u_{r*} - u_{l*}}{2\delta x}$$

$$u_{l*} = \left.\frac{\mathrm{d}y}{\mathrm{d}x}\right|_{l*} \approx \frac{y_n - y_l}{2\delta x}$$

$$u_{r*} = \left.\frac{\mathrm{d}y}{\mathrm{d}x}\right|_{r*} \approx \frac{y_r - y_n}{2\delta x}$$

$$\frac{\mathrm{d}u}{\mathrm{d}x} = \frac{\mathrm{d}^2 y}{\mathrm{d}x^2} \approx \frac{y_l - 2y_n + y_r}{(\Delta x)^2}$$



Finite differences: approximation errors

Finite differences can be deduced from Taylor's series. This is especially handy when analysing **approximation errors**.

For instance, employing a Taylor's expansion of $y(x)$ around x_n (to the right, i.e. at $y_r = y(x_r)$ with $x_r = x_n + \Delta x$):

$$y_r = y_n + \left.\frac{\mathrm{d}y}{\mathrm{d}x}\right|_{x_n} \frac{\Delta x^1}{1!} + \left.\frac{\mathrm{d}^2y}{\mathrm{d}x^2}\right|_{x_n} \frac{\Delta x^2}{2!} + \left.\frac{\mathrm{d}^3y}{\mathrm{d}x^3}\right|_{x_n} \frac{\Delta x^3}{3!} + \cdots$$

Expanding to the left (at $y_l = y(x_l)$ with $x_l = x_n - \Delta x$):

$$y_l = y_n - \left.\frac{\mathrm{d}y}{\mathrm{d}x}\right|_{x_n} \frac{\Delta x^1}{1!} + \left.\frac{\mathrm{d}^2y}{\mathrm{d}x^2}\right|_{x_n} \frac{\Delta x^2}{2!} - \left.\frac{\mathrm{d}^3y}{\mathrm{d}x^3}\right|_{x_n} \frac{\Delta x^3}{3!} + \cdots$$

Then, subtracting:

$$y_r - y_l = 2 \left.\frac{\mathrm{d}y}{\mathrm{d}x}\right|_{x_n} \frac{\Delta x^1}{1!} + 2 \left.\frac{\mathrm{d}^3y}{\mathrm{d}x^3}\right|_{x_n} \frac{\Delta x^3}{3!} + 2 \left.\frac{\mathrm{d}^5y}{\mathrm{d}x^5}\right|_{x_n} \frac{\Delta x^5}{5!} + \cdots$$

i.e., only odd derivative terms appear

Finite differences: approximation errors (cont.)

Solving for $\frac{\mathrm{d}y}{\mathrm{d}x}$:

$$\left.\frac{\mathrm{d}y}{\mathrm{d}x}\right|_{x_n} = \frac{y_r - y_l}{2\,\Delta x} \underbrace{- \left.\frac{\mathrm{d}^3y}{\mathrm{d}x^3}\right|_{x_n} \frac{\Delta x^2}{3!} - \left.\frac{\mathrm{d}^5y}{\mathrm{d}x^5}\right|_{x_n} \frac{\Delta x^4}{5!} - \cdots}_{\textbf{truncation error} \quad \Rightarrow \quad O(\Delta x^2)}$$

For $\Delta x \to 0$, the largest term on the truncation error (that can be controlled in the numerical method) is Δx^2. Therefore, we say that the central difference method is of **second order, quadratic, or** $O(\Delta x^2)$. For instance, if we reduce the step size by half, the error decreases 4 times.

Back to the first Taylor's expansion around x_n to the right, solving for $\frac{\mathrm{d}y}{\mathrm{d}x}$:

$$\left.\frac{\mathrm{d}y}{\mathrm{d}x}\right|_{x_n} = \frac{y_r - y_n}{\Delta x} \underbrace{- \left.\frac{\mathrm{d}^2y}{\mathrm{d}x^2}\right|_{x_n} \frac{\Delta x}{2!} - \left.\frac{\mathrm{d}^3y}{\mathrm{d}x^3}\right|_{x_n} \frac{\Delta x^2}{3!} + \cdots}_{\textbf{truncation error} \quad \Rightarrow \quad O(\Delta x)}$$

Finite differences: approximation errors (cont.)

Now, solving the Taylor's expansion to the left for $\frac{\mathrm{d}y}{\mathrm{d}x}$:

$$\left.\frac{\mathrm{d}y}{\mathrm{d}x}\right|_{x_n} = \frac{y_n - y_l}{\Delta x} + \underbrace{\left.\frac{\mathrm{d}^2 y}{\mathrm{d}x^2}\right|_{x_n} \frac{\Delta x}{2!} - \left.\frac{\mathrm{d}^3 y}{\mathrm{d}x^3}\right|_{x_n} \frac{\Delta x^2}{3!} + \cdots}_{\textbf{truncation error} \quad \Rightarrow \quad O(\Delta x)}$$

Therefore, although their truncation errors are different, both the forward and backward finite differences have approximation errors of first order. They are **first order, linear, or $O(\Delta x)$** methods. Thus, by reducing the step size by half, the error is expected to decrease by half as well (linearly).

Note that the three methods require information provided at just two grid points. By considering more points around x_n, higher order finite differences can be designed in an analogous manner. With more points the setting up of boundary conditions becomes more complicated though.

Finite differences in one dimension: Summary

At node i, with $h = \Delta x$, the forward difference is:

$$\boxed{\left.\frac{\partial y}{\partial x}\right|_{x_i} \approx \frac{y_{i+1} - y_i}{h}}$$

The backward difference is:

$$\boxed{\left.\frac{\partial y}{\partial x}\right|_{x_i} \approx \frac{y_i - y_{i-1}}{h}}$$

The central differences for first and second orders derivatives are:

$$\boxed{\left.\frac{\partial y}{\partial x}\right|_{x_i} \approx \frac{y_{i+1} - y_{i-1}}{2\,h}}$$

$$\boxed{\left.\frac{\partial^2 y}{\partial x^2}\right|_{x_i} \approx \frac{y_{i-1} - 2\,y_i + y_{i+1}}{h^2}}$$

Solution of initial value problems (IVPs) using finite differences

Euler/forward method (EF)

Let's start with the following initial value problem (IVP):

$$\frac{\mathrm{d}y}{\mathrm{d}x} = f(x,y) \quad \text{with} \quad y(x=x_0) = y_0$$

A forward finite difference approximation gives:

$$\frac{y_{i+1} - y_i}{h} = f_i \quad \text{with} \quad f_i = f(x_i, y_i)$$

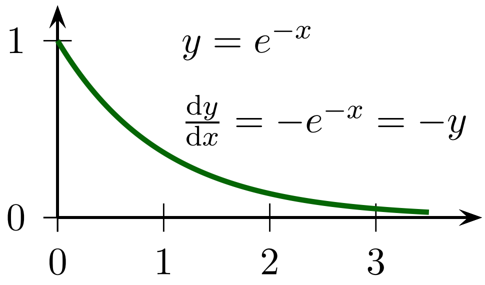
Thus, the resulting method, known as Euler/forward (EF) is:

$$\boxed{y_{i+1} = y_i + h\, f_i} \qquad \Rightarrow \qquad \textbf{explicit}$$

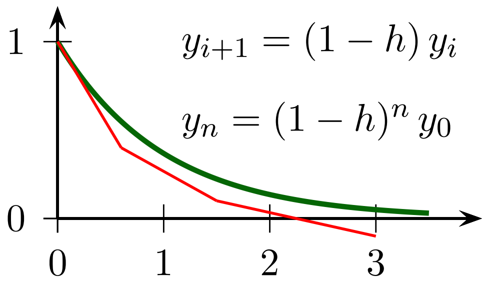
Note that the EF solution will incrementally update y_{i+1} with information **explicitly** based only at the current node: $f_i(x_i, y_i)$.

Euler/forward method (EF) (cont.)

For example, given the following IVP (with known closed-form solution):



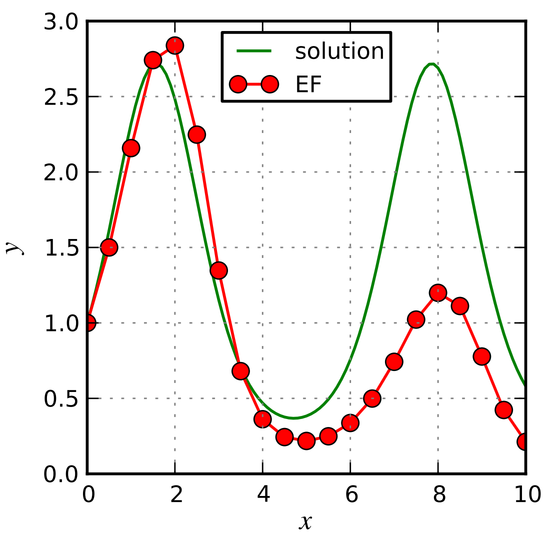
The EF solution is illustrated below:



Euler/forward method (EF): Python code

Fundamentals
1D FDM
Finite differences
Higher derivatives
Approx errors 1
Approx errors 2
Approx errors 3
Summary of schemes
FDM and IVPs
Euler/forward 1
Euler/forward 2
▷ EF: Python
Euler/backward 1
Euler/backward 2
EB: Python
EB EF comp
1D Stability
1D Poisson
1D Diffusion
1D Wave
2D Poisson
FDM Conclusions

The implementation of the EF method is very easy and straightforward (especially in Python). The code to the right may serve as an **example** an can be modified for other problems.



Solving $\frac{dy}{dx} = f(x, y) = y\, cos(x)$

```
from msys_fig import *
SetForEps(1.0,200) # for eps fig
def f(x,y): return y*cos(x)
def y(x):   return exp(sin(x))
x0, y0, xf = 0.0, 1.0, 10.0
x = linspace(x0,xf,101)
plot(x,y(x),'g-',label='solution')
h    = 0.5
x, y = x0, y0
X, Y = [x0], [y0]
while x < xf:
    if x + h > xf: h = xf - x
    y = Y[-1] + h * f(X[-1],Y[-1])
    x = X[-1] + h
    X.append(x)
    Y.append(y)
plot(X,Y,'ro',ls='-',label='EF')
Gll(r'$x$',r'$y$') # for eps fig
Save('efmethod1.eps') # eps fig
```

Euler/backward method (EB)

A backward finite difference approximation gives:

$$\frac{y_i - y_{i-1}}{h} = f_i \quad \text{with} \quad f_i = f(x_i, y_i)$$

Or, by adding 1 to i in the above expression:

$$\frac{y_{i+1} - y_i}{h} = f_{i+1}$$

Thus, the resulting method, known as Euler/backward (EB) is:

$$\boxed{y_{i+1} = y_i + h\, f_{i+1}} \qquad \Rightarrow \qquad \textbf{implicit}$$

A question arises: how can we find f_{i+1} to be used in this expression? Note we don't have y_{i+1} – it's precisely what we're after! The answer is: *let's start with a trial/guess value of y_{i+1}, let's say equal to y_i, and then apply the Newton-Raphson's method to solve:*

$$r(y_{i+1}) = y_{i+1} - y_i - h\, f_{i+1}$$

Euler/backward method (EB) (cont.)

With an initial guess y_{i+1}^k, where k stands for an **iteration** number, the equation to be solved is:

$$r(y_{i+1}^k) = y_{i+1}^k - y_i - h\, f_{i+1}^k$$

Thus, we want to find y_{i+1}^k such that $r(y_{i+1}^k)$ is zero. Therein, $f_{i+1}^k = f(x_{i+1}, y_{i+1}^k)$ is the corresponding trial value of $f(x, y)$. Applying Newton-Raphson's method:

$$r(y_{i+1}^k) + \left.\frac{\mathrm{d}r}{\mathrm{d}y}\right|_{y_{i+1}^k} \delta y = 0 \quad \Rightarrow \quad \delta y = -\left[\left.\frac{\mathrm{d}r}{\mathrm{d}y}\right|_{y_{i+1}^k}\right]^{-1} r(y_{i+1}^k)$$

Then, a better approximation of y_{i+1}^k is:

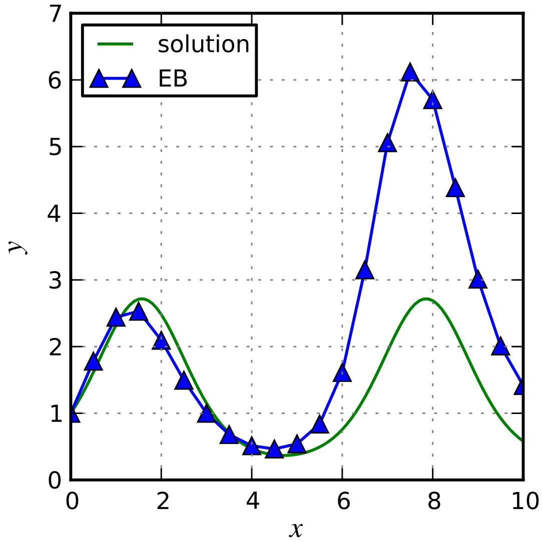
$$y_{i+1}^{k+1} = y_{i+1}^k + \delta y$$

Note that:

$$\left.\frac{\mathrm{d}r}{\mathrm{d}y}\right|_{y_{i+1}^k} = 1 - h \left.\frac{\mathrm{d}f}{\mathrm{d}y}\right|_{y_{i+1}^k} = 1 - h\, J(x_{i+1}, y_{i+1}^k)$$

Euler/backward method (EF): Python code

The implementation of the EB method is also quite simple (especially in Python). The code to the right may serve as an **example** for future modifications.



solving $\frac{dy}{dx} = f(x, y) = y\,cos(x)$

```
from msys_fig import *
SetForEps(1.0,200) # for eps fig
def f(x,y): return y*cos(x)
def J(x,y): return cos(x)
def y(x):   return exp(sin(x))
x0, y0, xf = 0.0, 1.0, 10.0
x = linspace(x0,xf,101)
plot(x,y(x),'g-',label='solution')
h    = 0.5
x, y = x0, y0
X, Y = [x0], [y0]
while x < xf:
    if x + h > xf: h = xf - x
    x = X[-1] + h
    y = Y[-1] # initial guess
    for it in range(10):
        r  = y - Y[-1] - h*f(x,y)
        if abs(r)<1.0e-10: break
        y += -r / (1.0 - h*J(x,y))
    X.append(x)
    Y.append(y)
plot(X,Y,'b^',ls='-',label='EB')
Gll(r'$x$',r'$y$') # for eps fig
Save('ebmethod1.eps') # eps fig
```


Euler methods (forward and backward): code comparison

Euler/forward:

```
def f(x,y): return y*cos(x)
# needs Jacobian   >>>>>>>>>>>>>>>>
x0, y0, xf = 0.0, 1.0, 10.0
h    = 0.5
x, y = x0, y0
while x < xf:
    if x + h > xf: h = xf - x
    x = X[-1] + h
    y = Y[-1] + h * f(X[-1],Y[-1])
    # needs iterations      >>>>>>
    # in order to find y    >>>>>>
    # since f now depends   >>>>>>
    # on new y.             >>>>>>
    # need to check errors  >>>>>>
```

Euler/backward:

```
def f(x,y): return y*cos(x)
def J(x,y): return cos(x)
x0, y0, xf = 0.0, 1.0, 10.0
h    = 0.5
x, y = x0, y0
while x < xf:
    if x + h > xf: h = xf - x
    x = X[-1] + h
    y = Y[-1] # initial guess
    for it in range(10):
        r  = y - Y[-1] - h*f(x,y)
        if abs(r)<1.0e-10: break
        y += -r / (1.0 - h*J(x,y))
    else: print 'convergence error'
```

Stability of Euler/forward (EF) and Euler/backward (EB) methods for initial value problems (IVPs)

Euler/forward method (EF): Stability

Test equation (Dahlquist's equation, see, e.g. *Hairer & Wanner, 1996, Solving Ordinary Differential Equattions I/II, Springer*), with $f(x, y) = \lambda\, y$:

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \lambda\, y \quad \text{with} \quad y_0 = 1 \quad \text{solution:} \quad y(x) = e^{\lambda\, x}$$

To asses the stability of the numerical method, we require that the exact solution $y(x)$ converges to a final value after large x. Thus, $\lambda < 0$ in this analysis (decay function). We also assume $h > 0$.

The EF approximation of this test equation is (with $f_i = \lambda\, y_i$):

$$\begin{aligned} y_{i+1} &= y_i + \lambda\, h\, y_i \\ &= (1 + \lambda\, h)\, y_i \end{aligned}$$

Hence, for any node n:

$$\boxed{y_n = (1 + \lambda\, h)^n\, y_0}$$

Euler/forward method (EF): Stability (cont.)

Now, after many steps ($n \to \infty$), the only way that the numerical solution y_n to our test equation can converge is when:

$$\begin{aligned} |1 + \lambda\, h| &\leq 1 \\ -1 \leq 1 + \lambda\, h &\leq 1 \\ -2 \leq \lambda\, h &\leq 0 \\ \lambda\, h \leq 0 \quad \text{and} \quad \lambda\, h &\geq -2 \end{aligned}$$

The first condition is already satisfied, since we assumed: $\lambda < 0$ and $h > 0$. The second one leads to:

$$\underbrace{-\lambda}_{|\lambda|}\; h \leq 2$$

thus:

$$\boxed{h \leq \frac{2}{|\lambda|}}$$

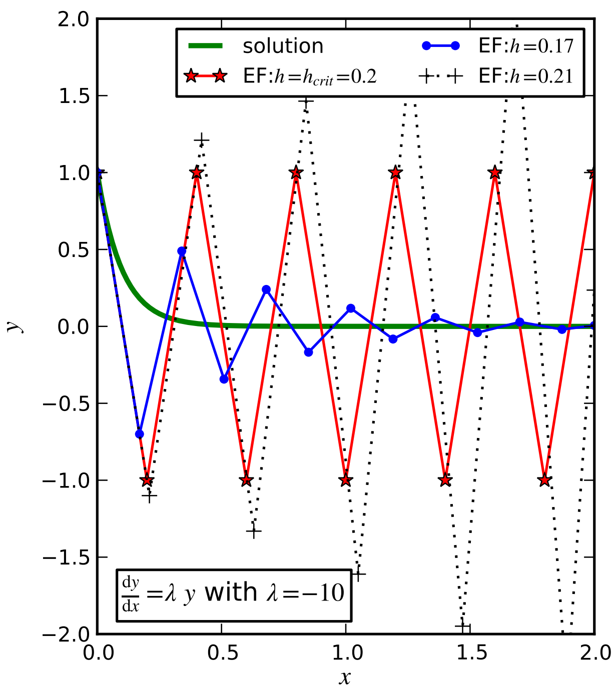
Euler/forward method (EF): Stability (cont.)

Therefore, for **stability**, h must be smaller than or equal to $h_{crit} = 2/|\lambda|$, i.e. the method is only **conditionally stable**.

For problems with larger $|\lambda|$, the step size h must be further reduced. Thus, for **stiff** problems ($|\lambda| \gg h$), the step size has to be even smaller in order to achieve stability.

Depending on h, the EF method may **converge** ($h < h_{crit}$), **diverge** ($h > h_{crit}$), or **oscillate** ($h = h_{crit}$) around the correct solution.

EF solution of $\frac{dy}{dx} = -10\, y$:



Euler/backward method (EB): Stability

Starting again with our test equation (Dahlquist's):

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \lambda\, y \quad \text{with} \quad y_0 = 1 \quad \text{solution:} \quad y(x) = e^{\lambda\, x}$$

The EB approximation of this test equation is (with $f_i = \lambda\, y_i$):

$$y_{i+1} = y_i + \lambda\, h\, y_{i+1}$$

Solving for y_{i+1}:

$$y_{i+1} = \frac{1}{1 - \lambda\, h}\, y_i$$

Hence, for any node n:

$$\boxed{y_n = \left(\frac{1}{1 - \lambda\, h}\right)^n y_0}$$

Euler/backward method (EB): Stability (cont.)

Again, we assume: $\lambda < 0$ and $h > 0$. Then, for stability, after many steps ($n \to \infty$):

$$\left|\frac{1}{1 - \lambda\, h}\right| \leq 1$$

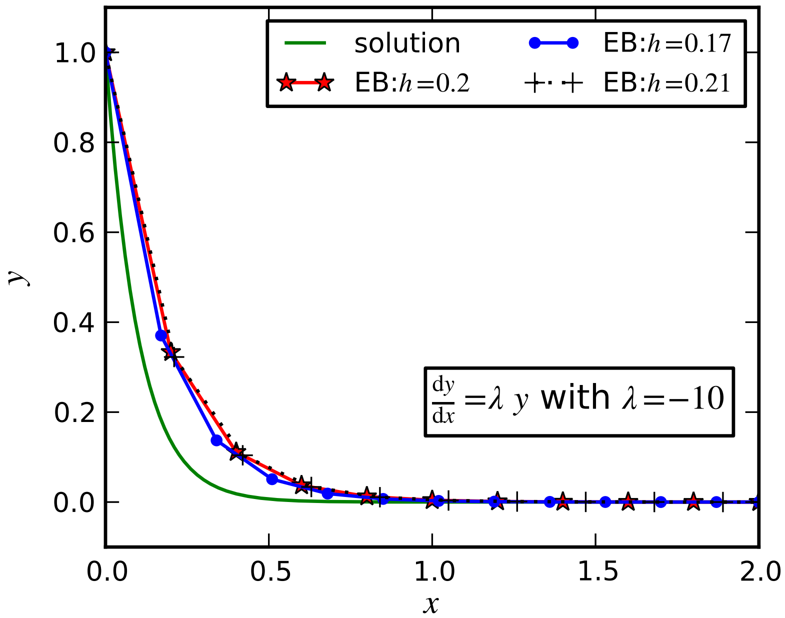
Note that $\left|\frac{1}{1-\lambda\, h}\right| \leq 1$ is always true, regardless the step size h, since $1 - \lambda\, h = 1 + |\lambda|\, h$ is always greater than or equal to 1.

Therefore, the EB method is **unconditionally stable**. This means that h can assume any value and the stability is not going to be affected. However, the **accuracy will**, of course, depend on h.

Euler/backward method (EB): Stability (cont.)

Fundamentals
1D FDM
1D Stability
EF: stability 1
EF: stability 2
EF: stability 3
EB: stability 1
EB: stability 2
▷ EB: stability 3
CE: stability 1
Stability: summary
1D Poisson
1D Diffusion
1D Wave
2D Poisson
FDM Conclusions

EB solution of Dahlquist's equation (compare with the previous EF solution) – this one is much more stable (no oscillations):



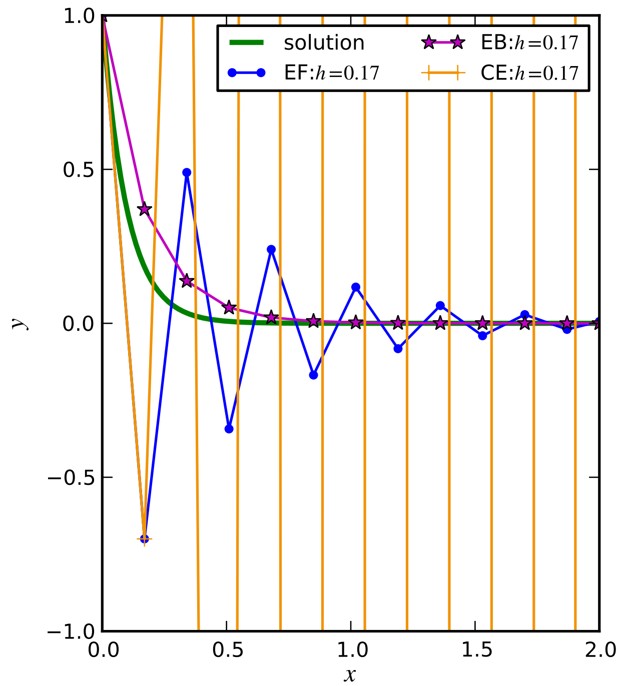
First order IVPs with central diff (not recommended)

Fundamentals
1D FDM
1D Stability
EF: stability 1
EF: stability 2
EF: stability 3
EB: stability 1
EB: stability 2
EB: stability 3
▷ CE: stability 1
Stability: summary
1D Poisson
1D Diffusion
1D Wave
2D Poisson
FDM Conclusions

Employing the same techniques and our model/test (Dahlquist) equation, it can be shown that Central Differences are **never stable** for first order IVBs: $\frac{dy}{dx} = f(x, y)$.

For example, when applied to $\frac{dy}{dx} = -10\, y$ with the same step sizes as for the Euler forward and backward methods, growing oscillations are obtained as shown in the figure to the right.

`CE` indicates central differences.



Euler's methods: summary on stability

- EF is **conditionally stable** and EB is **unconditionally stable**
- For first order IVPs, central differences are **unconditionally unstable**
- Unconditional stability is typical of implicit methods; the price we pay is the higher computational cost per step
- It is important to mention though that numerical stability **does not** imply accuracy
- For instance, the order of approximation error of the EB method is similar to the one computed with the EF method
- Thus, a "good" method needs to be **stable**, **accurate**, and **fast** (among some other properties, such as convergent, with uniqueness and robustness . . .)

Solution of Poisson's equation in 1D using finite differences

Finite difference approximation of Poisson's equation

In 1D (actually in 2D as well), this problem can be easily solved using finite differences. Given the 1D Poisson's equation (elliptic, BVP), where $\bar{x}$ indicates all points on boundaries:

$$-k_x \frac{\partial^2 u}{\partial x^2} = s(x) \quad \text{with} \quad \text{BCs:} \quad u(\bar{x}) = \bar{g}$$

Applying central differences of second order with a grid of N nodes; at each node i:

$$-k_x \frac{u_{i-1} - 2\,u_i + u_{i+1}}{\Delta x^2} = s(x_i)$$

Or, reorganising, with $s_i = s(x_i)$:

$$\underbrace{\frac{2\,k_x}{\Delta x^2}}_{\alpha} u_i \underbrace{-\frac{k_x}{\Delta x^2}}_{\beta} u_{i-1} \underbrace{-\frac{k_x}{\Delta x^2}}_{\beta} u_{i+1} = s_i$$

Thus, the solution is achieved by with of N equations where each one (i) is:

$$\boxed{\alpha\, u_i + \beta\, u_{i-1} + \beta\, u_{i+1} = s_i}$$

Finite difference approximation of Poisson's equation

In the finite differences literature, the coefficients α and β are known as *molecule* weights. These can be organised as follows:

$$m = \begin{Bmatrix} \alpha & \beta & \beta \end{Bmatrix}$$

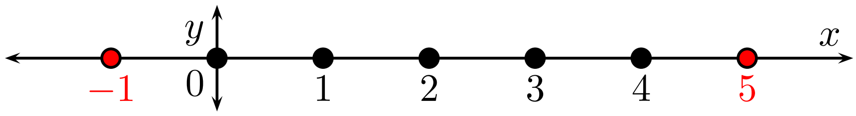
each corresponding to the three nodes around a particular node and itself. For example, a list with their indices can be defined as follows:

$$I = \begin{Bmatrix} i & i-1 & i+1 \end{Bmatrix}$$

After the finite differences discreteisation, the problem becomes a linear system with N equations. For example, with 5 nodes and $k_x = \Delta x = 1$, hence $\alpha = 2$ and $\beta = -1$, the system is:

$$\begin{array}{rrrrrrrl}
{\color{red}-u_{-1}} & +2\,u_0 & -u_1 & & & & & = s_0 \\
& -u_0 & +2\,u_1 & -u_2 & & & & = s_1 \\
& & -u_1 & +2\,u_2 & -u_3 & & & = s_2 \\
& & & -u_2 & +2\,u_3 & -u_4 & & = s_3 \\
& & & & -u_3 & +2\,u_4 & -{\color{red}u_5} & = s_4
\end{array}$$

In the previous system, note that the values u_{-1} and u_5 in red cannot be computed. To solve this, we have to introduce **boundary conditions**. Now, an assumption has to be made. First we employ the concept of *virtual* nodes:



Then, we assume that the first order gradients at the left and right boundaries are zero, i.e. $\frac{\partial u}{\partial x}\Big|_{\text{boundary}} = 0$. This means that we're assuming an **impermeable or insulated** condition at those points.

By setting the gradients at the boundary, and leaving them zero by **default**, we're setting the so-called **natural** boundary conditions.

Due to this impermeable boundary, for the left end, we use central finite differences as follows:

$$\left.\frac{\partial u}{\partial x}\right|_{x_0} \approx \frac{u_1 - u_{-1}}{2\,\Delta x} = 0 \qquad \Rightarrow \qquad u_{-1} = u_1 \quad \text{(mirrored)}$$

For the right end, also applying first order central differences:

$$\left.\frac{\partial u}{\partial x}\right|_{x_4} \approx \frac{u_5 - u_3}{2\,\Delta x} = 0 \qquad \Rightarrow \qquad u_5 = u_3 \quad \text{(mirrored)}$$

To make the computer implementation easier, the system to be solved can be written using matrices. For the hypothetical case discussed above:

$$\begin{bmatrix} 2 & -2 & & & \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ & & -1 & 2 & -1 \\ & & & -2 & 2 \end{bmatrix} \begin{Bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \end{Bmatrix} = \begin{Bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \end{Bmatrix}$$

where the components in red correspond to the **mirrored** ones.

Note that the coefficient matrix is sparse and tri-diagonal. Therefore, a specialised (efficient) solver should be used in order to obtain its solution. Nonetheless, here we're going to use a simple solver provided by NumPy/SciPy.

Finite difference approx. of Poisson's equation (cont.)

Because we have to specify *boundary values* in order to solve the corresponding BVP, we cannot solve the linear system just yet. For this reason, we cannot proceed in an incremental fashion as we did for IVPs. Therefore, numerical discretisations of BVPs usually lead to large linear systems of equations.

One kind of boundary condition is **essential**: the values of u at some points on the boundary. Example: u_0 and u_4 are **prescribed**. Then, we just need to remove the corresponding 0 and 4 equations from the linear system, after substituting for u_0 and u_4.

Nonetheless, to obtain a more **automatic** algorithm able to handle very large systems, we will first reorganise some equations by swapping rows and columns until the prescribed equations move to the "bottom" of the system. Remember that this swapping does not change the solution.

Finite difference approx. of Poisson's equation (cont.)

Let's first re-write the previous system with some components in colours:

$$\begin{bmatrix} 2 & -2 & & & \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ & & -1 & 2 & -1 \\ & & & -2 & 2 \end{bmatrix} \begin{Bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \end{Bmatrix} = \begin{Bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \end{Bmatrix}$$

Now, we move the first equation to the "bottom" and the first column to the right (this does not change the system):

$$\begin{bmatrix} 2 & -1 & & & -1 \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ & & -2 & 2 & \\ -2 & & & & 2 \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_0 \end{Bmatrix} = \begin{Bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_0 \end{Bmatrix}$$

Finite difference approx. of Poisson's equation (cont.)

And then split the system into two sub-systems. Therefore the $\boldsymbol{K}$ matrix is subdivided into four sub-matrices:

$$\left[\begin{array}{ccc|cc} 2 & -1 & & & -1 \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ \hline & & -2 & 2 & \\ -2 & & & & 2 \end{array}\right] \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_0 \end{Bmatrix} = \begin{Bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_0 \end{Bmatrix}$$

Or:

$$\begin{bmatrix} \boldsymbol{K_{11}} & \boldsymbol{K_{12}} \\ \boldsymbol{K_{21}} & \boldsymbol{K_{22}} \end{bmatrix} \begin{Bmatrix} \boldsymbol{U_1} \\ \boldsymbol{U_2} \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{F_1} \\ \boldsymbol{F_2} \end{Bmatrix}$$

With:

$$\boldsymbol{K_{11}} = \begin{bmatrix} 2 & -1 & \\ -1 & 2 & -1 \\ & -1 & 2 \end{bmatrix} \quad , \quad \boldsymbol{K_{12}} = \begin{bmatrix} & -1 \\ & \\ -1 & \end{bmatrix}$$

And:

$$\boldsymbol{K_{21}} = \begin{bmatrix} & & -2 \\ -2 & & \end{bmatrix} \quad , \quad \boldsymbol{K_{22}} = \begin{bmatrix} 2 & \\ & 2 \end{bmatrix}$$

Finite difference approx. of Poisson's equation (cont.)

In the previous equation: $\boldsymbol{U_2} = \begin{Bmatrix} u_4 & u_0 \end{Bmatrix}^T$ holds the known (or **prescribed**) values of u_0 and u_4.

Note that $\boldsymbol{K_{11}}$ is the only system we need to solve, after substituting the values of u_0 and u_4, and moving the corresponding terms to the right hand side.

Note also that the determinant of $\boldsymbol{K_{11}}$ is not zero anymore: $\det(\boldsymbol{K_{11}}) = 4$.

Since:

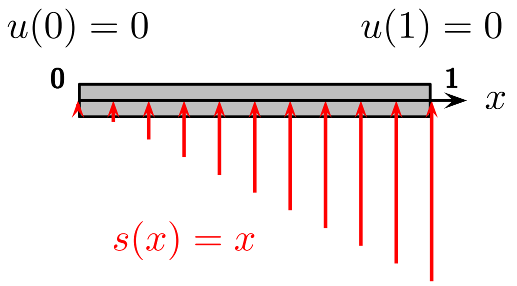
$$\boldsymbol{K_{11}} \boldsymbol{U_1} + \boldsymbol{K_{12}} \boldsymbol{U_2} = \boldsymbol{F_1}$$

Then:

$$\boxed{\boldsymbol{U_1} = \boldsymbol{K_{11}}^{-1} (\boldsymbol{F_1} - \boldsymbol{K_{12}} \boldsymbol{U_2})}$$

Thus, the unknown values u_1, u_2, and u_3, stored in $\boldsymbol{U_1} = \begin{Bmatrix} u_1 & u_2 & u_3 \end{Bmatrix}^T$, can now be found.

Example: A metallic rod is subject to an external (constant) heat source as illustrated below, where $u = T$ indicates the temperature. Both ends of this rod are kept at 0 units of temperature. The rod has a conductivity $k_x = 1$. Solve this steady heat problem for the temperature distribution along the rod.



Poisson's model equation:

$$-\frac{\partial^2 u}{\partial x^2} = x$$

with:

$$0 \leq x \leq 1$$

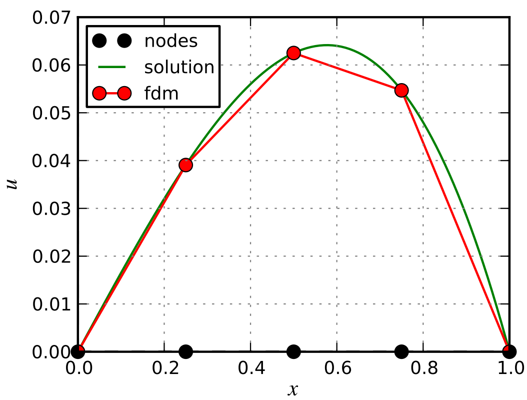
and:

$$u(x = 0) = 0$$
$$u(x = 1) = 0$$

The Python script to the right is used to solve this problem with 5 nodes. Note that the exact solution is $u(x) = \frac{x - x^3}{6}$.

The results, multiplied by 128 are given below.

```
[ 0.   5.   8.   7.   0.]
```




```
from msys_fig import *          # my fig routines
def s(x): return x              # source function
kx, N, dx = 1.0, 5, 0.25        # conduct, nx, dx
dxx = dx**2.0                   # dx squared
alp, bet = 2.0*kx/dxx, -kx/dxx # alpha, beta
mol = [alp, bet, bet]           # fdm molecule
K   = zeros((N,N))              # K matrix
F   = zeros(N)                  # F vector
for n in range(N):              # for each equat
    I = [n, n-1, n+1]           # neighbour nodes
    if n==0:   I[1] = I[2]      # left boundary
    if n==N-1: I[2] = I[1]      # right boundary
    for p, k in enumerate(I):   # each contrib
        K[n,k] += mol[p]        # set K matrix
    F[n] = s(n*dx)              # x = n*dx
U     = zeros(N)                # U vector
pn    = [0, N-1]                # prescribed nodes
U[pn] = [0.0, 0.0]              # prescribed vals
eqs   = arange(N)               # all equations
eq2   = eqs[pn]                 # prescribed eqs
eq1   = delete(eqs,eq2)         # eqs to be solved
K1_   = K  [eq1,:] # rows 1 of K, any column
K11   = K1_[:,eq1] # any row, cols 1 of K1_
K12   = K1_[:,eq2] # any row, cols 2 of K1_
U[eq1] = solve(K11, F[eq1] - dot(K12, U[eq2]))
print U * 128.                  # print results
```

Solution of the diffusion equation in 1D using finite differences

Finite difference approximation of the diffusion equation

Given the 1D diffusion equation (parabolic, IBVP), where $\bar{x}$ indicates all "points" on boundaries:

$$\frac{\partial u}{\partial t} - k_x \frac{\partial^2 u}{\partial x^2} = s(x) \quad \text{with} \quad \begin{array}{ll} \text{ICs:} & u(t=0,x) = u_0(x) \\ \text{BCs:} & u(\bar{x},t) = \bar{u}(t) \end{array}$$

Applying central differences of second order to $\frac{\partial^2 u}{\partial x^2}$, with a grid of N nodes, at each node i:

$$\left.\frac{\partial u}{\partial t}\right|_{x_i} = s(x_i) + \frac{k_x}{\Delta x^2}(u_{i-1} - 2\,u_i + u_{i+1})$$

Thus, the space coordinate x has been discretised while the time coordinate t has not. We say that the above system of equations is a **semi-discrete** form of the corresponding IBVP. This is known as the **method of lines**. The **B** part of the problem has been discretised. Now, another numerical method has to be employed in order to solve the **I** part of the problem. Thus, we're left with an IVP system to be solved, with as many equations as N.

Finite diff. approx. of the diffusion equation (cont.)

Note that the second term on the right-hand side of the previous equation (semi-discretised diffusion equation) will produce a matrix similar to the one discussed when applying central differences to Poisson's equation. Nonetheless, in this case, a matrix $\boldsymbol{K}$ does not need to be formed, since we can progressively solve for the time variable with an IVP solver.

The IVP to be solved can be represented as follows:

$$\frac{\partial \boldsymbol{U}}{\partial t} = f(t, \boldsymbol{U}) \quad \text{with} \quad f_i(t, \boldsymbol{U}) = s_i + \frac{k_x}{\Delta x^2}(u_{i-1} - 2\,u_i + u_{i+1})$$

where:

$$\boldsymbol{U} = \begin{bmatrix} u_0 & u_1 & u_2 & \dots & u_N \end{bmatrix}^T$$

This is a system of ODEs. Hence, similar stability analyses as the ones we did before for IVPs should be made here. For instance, explicit methods might be **conditionally stable**, where the size of Δt (time step) will depend on some conditions.

Finite diff. approx. of the diffusion equation (cont.)

On the other hand, implicit schemes may be **unconditionally stable** (any Δt should not cause instabilities; accuracy might be lost with large values though).

Because our IVP is now a (large) system, stability analyses require a little more effort in order to be deduced. Nonetheless, by means of the eigenvalues of the right-hand side matrix (without the source term) and Fourier series, analyses of stability can be carried out.

For example, if we employ the Euler/forward method for the solution in time:

$$\frac{u_{j+1} - u_i}{\Delta t} = s_i + \frac{k_x}{\Delta x^2}(u_{i-1} - 2\,u_i + u_{i+1})$$

$$u_{j+1} = u_i + s_i + \frac{k_x\,\Delta t}{\Delta x^2}(u_{i-1} - 2\,u_i + u_{i+1})$$

It can be shown that stability requires that:

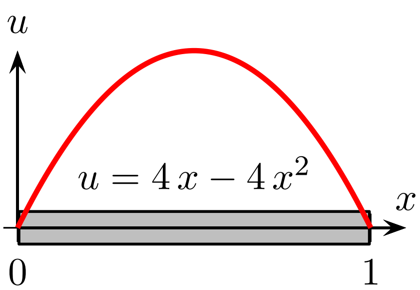
$$R = \frac{k_x\,\Delta t}{\Delta x^2} \leq \frac{1}{2} \qquad R \text{ is known as } \textit{diffusion} \text{ number}$$

The computer implementation of the semi-discrete solution to the diffusion equation is quite straightforward. For the solution in time, any ODE solver could be used.

Here, we'll use the Euler/forward one. Its implementation in Python is given to the right, where a slightly modified version is shown in order to account for **system** of ODEs, e.g. now `y0` can be an array. Therefore, list `Y` is now a list of lists, where each sublist corresponds to one component of `y0`.

```
# Copyright 2012
Dorival de Moraes Pedroso. All rights reserved.
# Use of this source code is governed by a BSD-
# license that can be found in the LICENSE file

def efsolve(x0,y0,xf,h,f):
    x, y, n = x0, y0, len(y0)
    X, Y = [x0], []
    for i in range(n):
        Y.append([y0[i]])
    while x < xf:
        if x + h > xf: h = xf - x
        y += h * f(x, y)
        x += h
        X.append(x)
        for i in range(n):
            Y[i].append(y[i])
    return X, Y
```

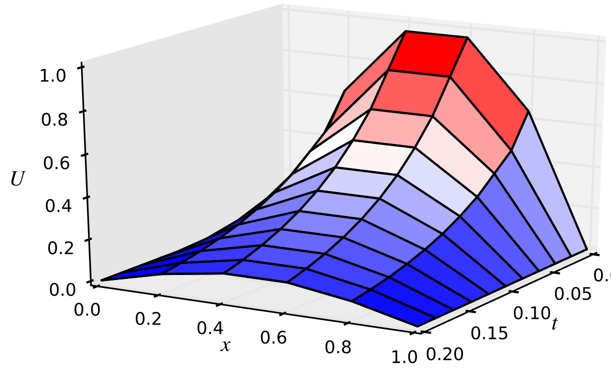

Example: let's solve the cooling down of an insulated wire of unit length, where only the two extremes are exposed to a temperature equal to 0. The conductivity used is 1. For some hypothetical reason, the wire had an initial temperature u distributed as below:



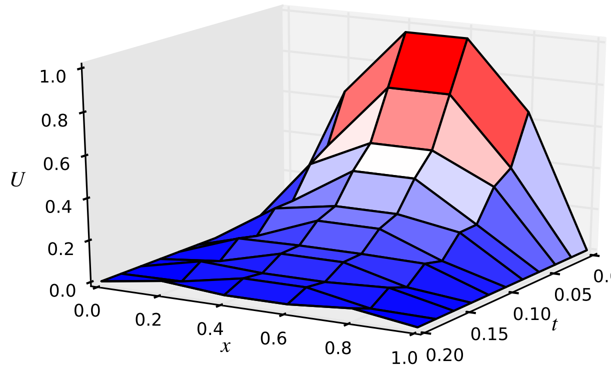

```
from msys_fig    import *    # my drawing routines
from efmethodvec import *    # Euler/forward method
kx    = 1.0                  # conductivity
N     = 6                    # number of nodes
xmax  = 1.0                  # length
dx    = xmax/float(N-1)      # spatial step size
X     = linspace(0.,xmax,N)  # grid nodes
U     = 4.0*X-4.0*X**2.0     # initial values
pn    = [0, N-1]             # prescribed nodes
U[pn] = [0.0, 0.0]           # prescribed values
m     = [1.0, -2.0, 1.0]     # fdm molecule
def f(t,U):                  # dUdt = f(t,U)
    dUdt = zeros(N)          # rate of U
    for i in range(N):       # for each node
        if i in pn: continue # skip if presc node
        for p, j in enumerate([i-1, i, i+1]):
            if j<0:  j = i+1 # left boundary
            if j==N: j = i-1 # right boundary
            dUdt[i] += kx*m[p]*U[j]/(dx**2.0)
    return dUdt # return dUdt at current (t,U)
t0, tf, dt = 0.0, 0.2, 0.02
print 'diff num R =', kx*dt/(dx**2.0)
T, Uxt = efsolve(t0, U, tf, dt, f)
tt,xx  = meshgrid(T,X)
ax = PlotSurf(tt,xx,vstack(Uxt),'t','x','U',0.,1.)
```

Fundamentals
1D FDM
1D Stability
1D Poisson
1D Diffusion
FDM: Diffusion 1
FDM: Diffusion 2
FDM: Diffusion 3
FDM: Diffusion 4
FDM: Diffusion 5
▷ FDM: Diffusion 6
FDM: Diffusion 7
1D Wave
2D Poisson
FDM Conclusions

The solution is presented in the figure to the right (top), with $k_x = 1$ and $R = 0.5$. We can see that the temperature decreases (from red to blue) along the t (time) axis.

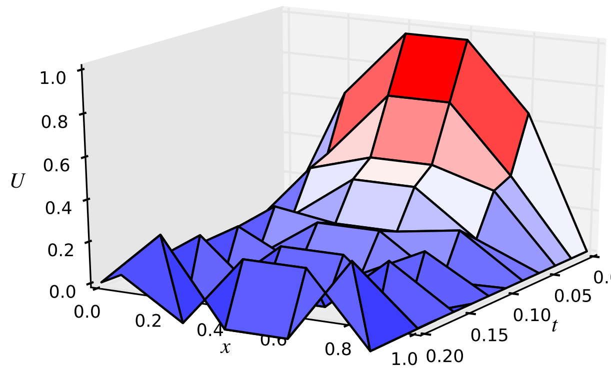


With $k_x = 1.5$ and $R = 0.75$, i.e. above the stability limit of 0.5, we can observe some noise: figure to the right (bottom).

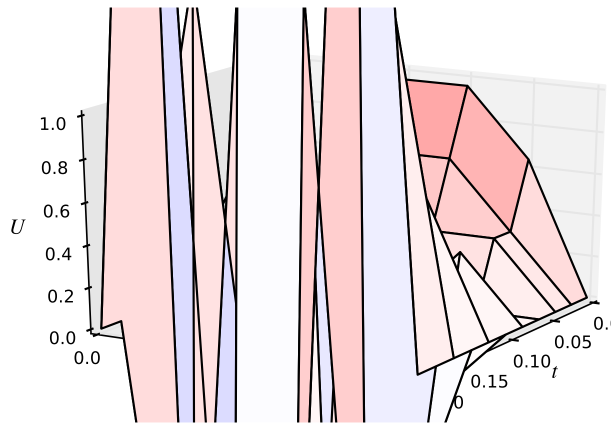


Fundamentals
1D FDM
1D Stability
1D Poisson
1D Diffusion
FDM: Diffusion 1
FDM: Diffusion 2
FDM: Diffusion 3
FDM: Diffusion 4
FDM: Diffusion 5
FDM: Diffusion 6
▷ FDM: Diffusion 7
1D Wave
2D Poisson
FDM Conclusions

By increasing k_x to 1.7 and keeping the same $\Delta t = 0.02$ and $\Delta x = 0.2 \Rightarrow R = 0.85$, the solution becomes more unstable (see fig to the right/top).



With $k_x = 2.0$ and $R = 1.0$, i.e. twice more than the stability limit of 0.5, the calculation "blows" up (see fig to the right/bottom).



Solution of the wave equation in 1D using finite differences

Finite difference approximation of the Wave equation

We aim to obtain an approximated solution to the 1D wave equation (hyperbolic, IBVP). This equation can represent, for example, the transversal vibration of a string or the longitudinal vibration of a rod with one fixed end.

In the following equation, $\bar{\rho} = \rho\, A$ represents the density per unit length and τ can be either the tension on the string or a stiffness parameter (modulus) of the rod:

$$\rho\,\frac{\partial^2 u}{\partial t^2} - \tau\,\frac{\partial^2 u}{\partial x^2} = s(x) \quad \text{with} \quad \text{ICs:} \begin{cases} u(t=0,x) & = u_0(x) \\ \left.\frac{\partial u}{\partial x}\right|_{t=0} & = v_0(x) \end{cases}$$

In the same way as we did for the diffusion equation, we apply central differences of second order to the $\frac{\partial^2 u}{\partial x^2}$ term. Thus, we get a semi-discrete system of equations where N is the number of nodes and i indicates a specific node:

$$\left.\frac{\partial^2 u}{\partial t^2}\right|_{x_i} = \frac{s}{\rho} + \frac{\tau}{\rho\,\Delta x^2}(u_{i-1} - 2\,u_i + u_{i+1})$$

Finite difference approx. of the Wave equation (cont.)

To solve this system we need 4 conditions: 2 initial values and 2 boundary values. The boundary values can be directly specified at the FDM nodes. Now, we still have to apply some scheme for the time derivative term $\frac{\partial^2 u}{\partial t^2}$. One approach is to employ again central finite differences. In doing so, we get the so-called **Leapfrog method**. Another approach, is to consider the first time derivative of u (the transversal velocity of a point on the string). Then, we will have to solve 2 systems of equations, one for u and another for $v = \frac{\partial u}{\partial t}$ (velocity). In fact, a system of $2\,N$ equations has to be solved.

This system is an initial value problem and hence the Euler methods discussed earlier can be employed. For its solution, initial conditions must be specified – actually 2 sets, one for the positions u_i and another for v_i. If we group (stack) these two vectors, the IVP problem can be symbolised by:

$$\frac{\mathrm{d}\boldsymbol{Y}}{\mathrm{d}t} = f(t, \boldsymbol{Y}) \quad \text{with} \quad \boldsymbol{Y} = \begin{bmatrix}\boldsymbol{U} & \boldsymbol{V}\end{bmatrix}^T$$

Finite difference approx. of the Wave equation (cont.)

Let's first define the well known **wave velocity** c as follows:

$$c = \sqrt{\frac{\mathcal{T}}{\rho}}$$

Also, let's define:

$$F_i = \frac{s_i}{\rho} + \frac{c^2}{\Delta x^2}(u_{i-1} - 2\,u_i + u_{i+1})$$

Then, the IVP to be solved is:

$$\frac{\mathrm{d}\boldsymbol{Y}}{\mathrm{d}t} = f(t, \boldsymbol{Y})$$

With:

$$\frac{\mathrm{d}\boldsymbol{Y}}{\mathrm{d}t} = \begin{Bmatrix}\frac{\mathrm{d}\boldsymbol{U}}{\mathrm{d}t} \\ \frac{\mathrm{d}\boldsymbol{V}}{\mathrm{d}t}\end{Bmatrix} \quad \text{and} \quad f(t, \boldsymbol{Y}) = \begin{Bmatrix}\boldsymbol{V} \\ \boldsymbol{F}\end{Bmatrix}$$

When applying the Euler/forward method, for instance, F_i gets multiplied by the time step Δt and the following coefficient can be defined:

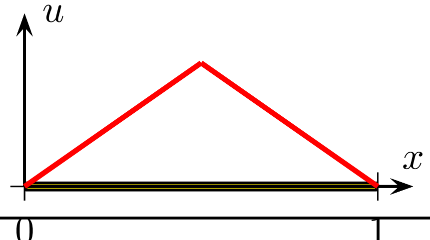
$$R = \frac{c^2\,\Delta t}{\Delta x^2}$$

To get a solution with the Euler/forward method, the same stability conditions apply. In this case, to guarantee stability, the following condition must be satisfied:

$$R \leq \frac{1}{2}$$

Example: let's find the deformation of a string fixed at its two extremes, after a transversal pull has been applied. The string has unit length $L = 1$, linear density $\rho = 1$, and is under a tension of $\tau = 1$. The initial position $u(t=0,x) = u_0$ is given by:

$$u_0 = \begin{cases} x & \text{if} \quad x < 0.5 \\ 1 - x & \text{otherwise} \end{cases}$$



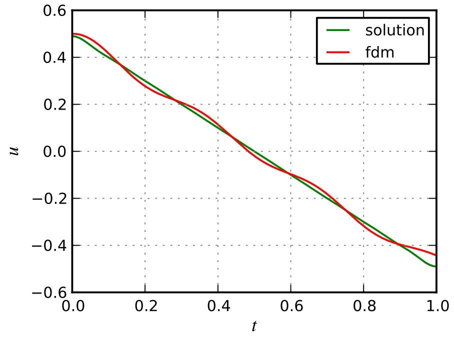

```
from msys_fig    import *     # my drawing routines
from efmethodvec import *     # Euler/forward method
from string_sol  import *     # analytical solution
SetForEps(0.75,250)           # for eps figure
tau = 1.0                     # tension
rho = 1.0                     # linear density
cc  = tau/rho                 # sq of wave speed
L   = 1.0                     # length of string
N   = 11                      # number of points
dx  = L/float(N-1)            # spatial step size
X   = linspace(0.,L,N)        # grid nodes
u0  = vectorize(lambda x: x if x<0.5 else 1.-x)
U   = u0(X)                   # initial values: displ
V   = zeros(N)                # initial values: veloc
pn  = [0, N-1]                # prescribed nodes
m   = [1.0, -2.0, 1.0]        # fdm molecule
```

(continued)

Finite difference approx. of the Wave equation (cont.)

Fundamentals
1D FDM
1D Stability
1D Poisson
1D Diffusion
1D Wave
FDM: Wave 1
FDM: Wave 2
FDM: Wave 3
FDM: Wave 4
FDM: Wave 5
▷ FDM: Wave 6
FDM: Wave 7
2D Poisson
FDM Conclusions

The displacement of a point at the middle of the string is shown below for times from 0 to 1:

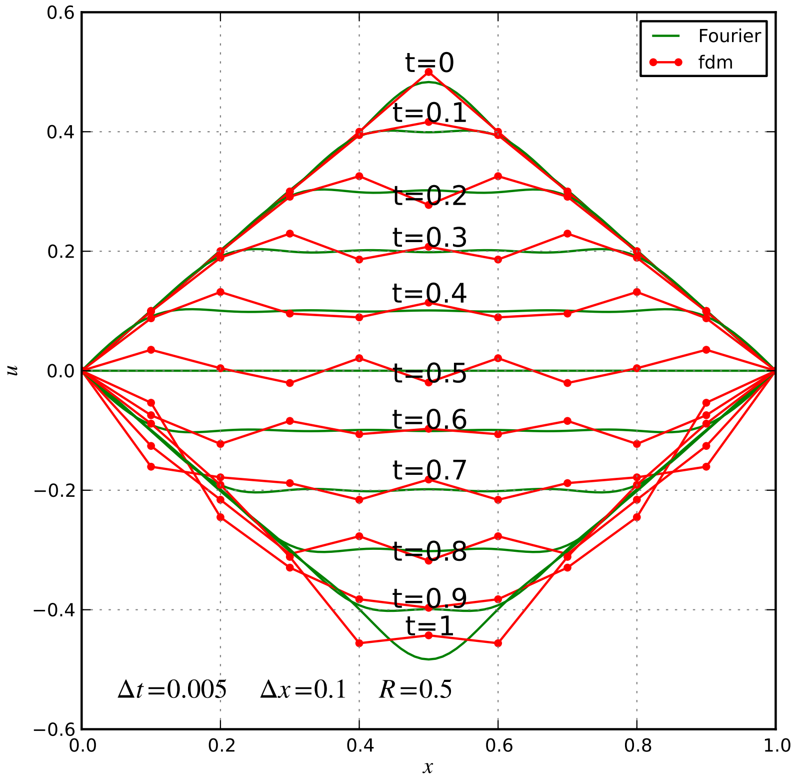


The position of all points for a number of selected times are shown in the next slide.

```
def f(t,Y):                      # dYdt = f(t,Y)
    U    = Y[:N]                 # unpack U
    V    = Y[N:]                 # unpack V
    dUdt = zeros(N)              # rate of U
    dVdt = zeros(N)              # rate of V
    for i in range(N):           # for each node
        if i in pn: continue     # skip if presc node
        dUdt[i] = V[i]           # rate of displ = vel
        for p, j in enumerate([i-1, i, i+1]):
            dVdt[i] += cc*m[p]*U[j]/(dx**2.0)
    return hstack((dUdt,dVdt))  # dYdt at current (t,Y)
t0, tf, dt = 0.0, 1.0, 0.005
print 'CFL =', cc*dt/(dx**2.0)
Y    = hstack((U,V))                 # pack U and V
T, Yxt = efsolve(t0, Y, tf, dt, f)  # update time
Uxt = Yxt[:N]                        # unpack U
plot(T,usol(sqrt(cc),L,T,0.5),'g-',label='solution')
plot(T,Uxt[5],'r-',label='fdm')      # fdm solution
Gll(r'$t$',r'$u$')                   # labels and grid
Save('wave1d_string.eps')            # save eps fig
```


Finite difference approx. of the Wave equation (cont.)

Fundamentals
1D FDM
1D Stability
1D Poisson
1D Diffusion
1D Wave
FDM: Wave 1
FDM: Wave 2
FDM: Wave 3
FDM: Wave 4
FDM: Wave 5
FDM: Wave 6
▷ FDM: Wave 7
2D Poisson
FDM Conclusions

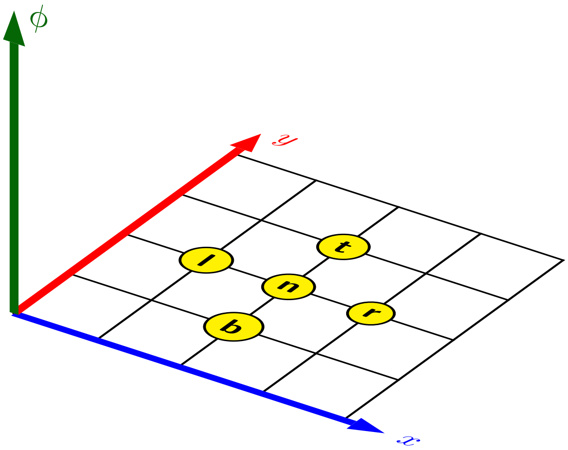


Solution of Poisson's equation in 2D using finite differences

Finite difference approximation of Poisson's eq in 2D

Finite differences can be easily applied to multi-dimensional problems as well. To do so, each partial derivative is approximated as we did for the 1D cases. For instance, suppose we have a function $\phi(x, y)$ that we want so solve for. It can be drawn in the 3D grid illustrated to the right.

We then write the approximation of any derivative evaluated at a point **n**. For the x direction we use the indices: **l** (left) and **r** (right). For the y direction, we use the indices: **b** (bottom) and **t** (top).



For first order derivatives:

$$\left.\frac{\partial \phi}{\partial x}\right|_{(x_n, y_n)} \approx \frac{\phi_r - \phi_l}{2\Delta x}$$

$$\left.\frac{\partial \phi}{\partial y}\right|_{(x_n, y_n)} \approx \frac{\phi_t - \phi_b}{2\Delta y}$$

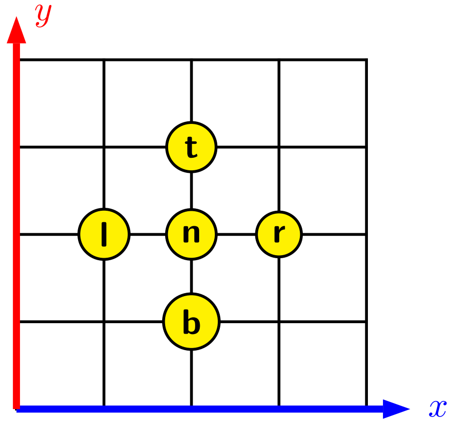
Applying central differences to second order derivatives:

$$\left.\frac{\partial^2 \phi}{\partial x^2}\right|_{(x_n, y_n)} \approx \frac{\phi_l - 2\phi_n + \phi_r}{(\Delta x)^2}$$

$$\left.\frac{\partial^2 \phi}{\partial y^2}\right|_{(x_n, y_n)} \approx \frac{\phi_b - 2\phi_n + \phi_t}{(\Delta y)^2}$$

Now, to solve any IBVP, initial and boundary conditions must be prescribed. In this notes, boundary conditions will be set in such a way that all x-y boundaries will have a zero **first order** gradient, i.e., boundaries are **impemeable** of **insulated** by default.

Looking from the above of the x-y grid:



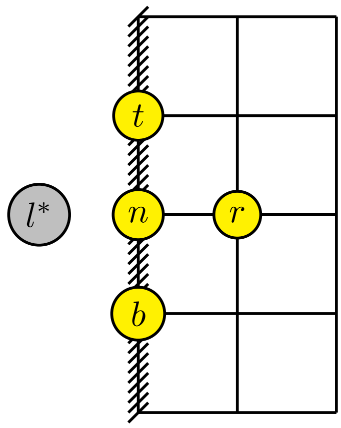
The way we set boundary conditions up is usually classified among **Dirichlet or Essential (DE)** and **Neumann of Natural (NN)**.

To memorize: DE is *essentially* required to solve **D**ifferntial **E**quations. NN arises *naturally* when specifying gradients (such as insulated boundaries).

Since we will assume by **default** that all boundaries are impermeable:

$$\left.\frac{\partial \phi}{\partial x}\right|_{boundary} = 0$$

$$\left.\frac{\partial \phi}{\partial y}\right|_{boundary} = 0$$



We can then employ central finite differences and a method that considers **virtual** nodes:

$$\left.\frac{\partial \phi}{\partial x}\right|_{bou.} = \frac{\phi_r - \phi_{l^*}}{2\Delta x} = 0$$

Thus: $\boxed{\phi_{l^*} = \phi_r}$

Fundamentals
1D FDM
1D Stability
1D Poisson
1D Diffusion
1D Wave
2D Poisson
FDM: Poisson 1
FDM: Poisson 2
FDM: Poisson 3
▷ FDM: Poisson 4
FDM: Poisson 5
FDM: Poisson 6
FDM: Poisson 7
FDM: Poisson 8
FDM: Poisson 9
FDM: Poisson 10
FDM: Poisson 11
FDM: Poisson 12
FDM: Poisson 13
FDM Conclusions

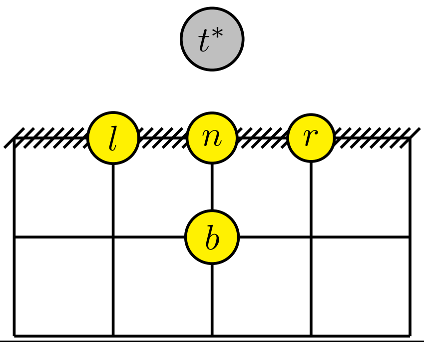
Applying the idea of **virtual** nodes to the y direction:

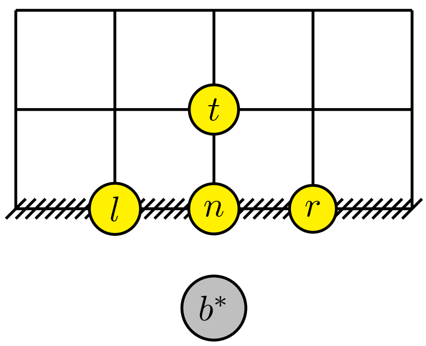
$$\left.\frac{\partial \phi}{\partial y}\right|_{bou.} = \frac{\phi_t - \phi_{b^*}}{2\Delta y} = 0$$

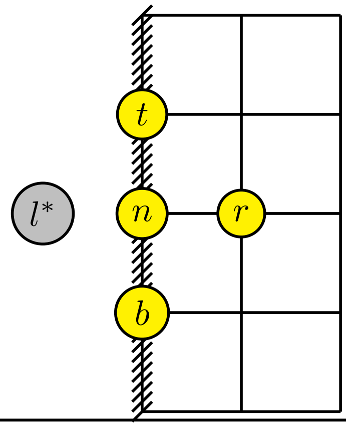
Thus: $\boxed{\phi_{b^*} = \phi_t}$

The same can be done for the right and top boundaries, leading to:

$\boxed{\phi_{r^*} = \phi_l}$ and $\boxed{\phi_{t^*} = \phi_b}$







Fundamentals
1D FDM
1D Stability
1D Poisson
1D Diffusion
1D Wave
2D Poisson
FDM: Poisson 1
FDM: Poisson 2
FDM: Poisson 3
FDM: Poisson 4
▷ FDM: Poisson 5
FDM: Poisson 6
FDM: Poisson 7
FDM: Poisson 8
FDM: Poisson 9
FDM: Poisson 10
FDM: Poisson 11
FDM: Poisson 12
FDM: Poisson 13
FDM Conclusions

Now, employing central differences for second order derivatives, for every node on the (impermeable) boundary:

$$\left.\frac{\partial^2 \phi}{\partial x^2}\right|_{n\text{(left/bou.)}} = \frac{\phi_{l^*} - 2\phi_n + \phi_r}{\Delta x^2} = \frac{\phi_r - 2\phi_n + \phi_r}{\Delta x^2} = \frac{-2\phi_n + 2\phi_r}{\Delta x^2}$$

For every node to the right boundary:

$$\left.\frac{\partial^2 \phi}{\partial x^2}\right|_{n\text{(right/bou.)}} = \frac{\phi_l - 2\phi_n + \phi_{r^*}}{\Delta x^2} = \frac{\phi_l - 2\phi_n + \phi_l}{\Delta x^2} = \frac{-2\phi_n + 2\phi_l}{\Delta x^2}$$

For every node to the bottom boundary:

$$\left.\frac{\partial^2 \phi}{\partial y^2}\right|_{n\text{(bott/bou.)}} = \frac{\phi_{b^*} - 2\phi_n + \phi_t}{\Delta y^2} = \frac{\phi_t - 2\phi_n + \phi_t}{\Delta y^2} = \frac{-2\phi_n + 2\phi_t}{\Delta y^2}$$

For every node to the top boundary:

$$\left.\frac{\partial^2 \phi}{\partial y^2}\right|_{n\text{(top/bou.)}} = \frac{\phi_b - 2\phi_n + \phi_{t^*}}{\Delta y^2} = \frac{\phi_b - 2\phi_n + \phi_b}{\Delta y^2} = \frac{-2\phi_n + 2\phi_b}{\Delta y^2}$$

We aim to obtain an approximated solution to the 2D Poisson's equation (elliptic, BVP). This equation can represent, for example, the heat through a metallic plate or the water seepage through a porous medium:

$$-k_x \frac{\partial^2 u}{\partial x^2} - k_y \frac{\partial^2 u}{\partial y^2} = s(x, y)$$

Applying central finite differences to both second order differential terms (disregarding boundary nodes at first):

$$-\frac{k_x}{\Delta x^2}(u_l - 2\,u_n + u_r) - \frac{k_y}{\Delta y^2}(u_b - 2\,u_n + u_t) = s_n$$

Thus, organising:

$$\underbrace{2\left(\frac{k_x}{\Delta x^2} + \frac{k_y}{\Delta y^2}\right)}_{\alpha} u_n \underbrace{-\frac{k_x}{\Delta x^2}}_{\beta} u_l \underbrace{-\frac{k_x}{\Delta x^2}}_{\beta} u_r \underbrace{-\frac{k_y}{\Delta y^2}}_{\gamma} u_b \underbrace{-\frac{k_y}{\Delta y^2}}_{\gamma} u_t = s_n$$

Thus, a linear system with as many equations as the number of nodes has to be solved. First, of course, the known values at the boundaries must be removed from this system.

Each equation, corresponding to a node **n**, is:

$$\boxed{\alpha\, u_n + \beta\, u_l + \beta\, u_r + \gamma\, u_b + \gamma\, u_t = s_n}$$

Employing the method of virtual nodes, the equations for nodes on boundaries are:

$$\begin{array}{llllll}
\text{left:} & \alpha\, u_n & & +2\,\beta\, u_r & +\gamma\, u_b & +\gamma\, u_t & = s_n \\
\text{right:} & \alpha\, u_n & +2\,\beta\, u_l & & +\gamma\, u_b & +\gamma\, u_t & = s_n \\
\text{bott.:} & \alpha\, u_n & +\beta\, u_l & +\beta\, u_r & & +2\,\gamma\, u_t & = s_n \\
\text{top.:} & \alpha\, u_n & +\beta\, u_l & +\beta\, u_r & +2\,\gamma\, u_b & & = s_n
\end{array}$$

In the finite differences literature, the coefficients α, β, γ are known as *molecule* weights. These can be organised as follows:

$$m = \{\alpha \quad \beta \quad \beta \quad \gamma \quad \gamma\}$$

Finite difference approx. of Poisson's eq in 2D (cont.)

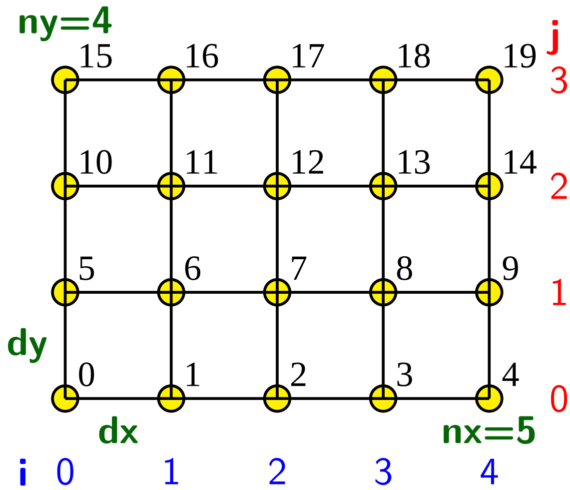
Fundamentals
1D FDM
1D Stability
1D Poisson
1D Diffusion
1D Wave
2D Poisson
FDM: Poisson 1
FDM: Poisson 2
FDM: Poisson 3
FDM: Poisson 4
FDM: Poisson 5
FDM: Poisson 6
FDM: Poisson 7
▷ FDM: Poisson 8
FDM: Poisson 9
FDM: Poisson 10
FDM: Poisson 11
FDM: Poisson 12
FDM: Poisson 13
FDM Conclusions

Now a loop can be developed in which each equation is set with the molecule array m. To account for insulated boundary conditions (natural/Neumann), the corresponding molecule can be changed by correcting those doubled coefficients.

To help with a computer implementation, a **class** can be developed to hold all data corresponding to a rectangular grid (n_x, n_y).

For example, **i** is used to loop over columns and **j** to loop over rows. Each node can then be easily found with the following (**integer**) expressions:

$$n = i + j\,n_x$$
$$l = n - 1 \qquad r = n + 1$$
$$b = n - n_x \qquad t = n + n_x$$
$$i = n \,\%\, n_x \qquad j = n \,/\, n_x$$

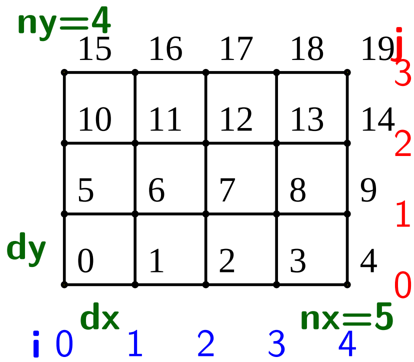


Finite difference approx. of Poisson's eq in 2D (cont.)

Fundamentals
1D FDM
1D Stability
1D Poisson
1D Diffusion
1D Wave
2D Poisson
FDM: Poisson 1
FDM: Poisson 2
FDM: Poisson 3
FDM: Poisson 4
FDM: Poisson 5
FDM: Poisson 6
FDM: Poisson 7
FDM: Poisson 8
▷ FDM: Poisson 9
FDM: Poisson 10
FDM: Poisson 11
FDM: Poisson 12
FDM: Poisson 13
FDM Conclusions

For instance, looping over every equation to pick the right molecule component I:

```
[0, 1, 1, 5, 5]
[1, 0, 2, 6, 6]
[2, 1, 3, 7, 7]
[3, 2, 4, 8, 8]
[4, 3, 3, 9, 9]
[5, 6, 6, 0, 10]
[6, 5, 7, 1, 11]
[7, 6, 8, 2, 12]
[8, 7, 9, 3, 13]
[9, 8, 8, 4, 14]
[10, 11, 11, 5, 15]
[11, 10, 12, 6, 16]
[12, 11, 13, 7, 17]
[13, 12, 14, 8, 18]
[14, 13, 13, 9, 19]
[15, 16, 16, 10, 10]
[16, 15, 17, 11, 11]
[17, 16, 18, 12, 12]
[18, 17, 19, 13, 13]
[19, 18, 18, 14, 14]
```




```
from grid2d import *                 # my 2d grid
Lx, Ly = 4.0, 3.0                    # lengths
g      = Grid2D(Lx,Ly,5,4)           # 5x4 grid
N      = g.nx * g.ny                 # num of nodes
for n in range(N):                   # for each node
    i, j = n%g.nx, n/g.nx            # col and rows
    I = [n, n-1, n+1, n-g.nx, n+g.nx] # nodes
    if i==0:      I[1] = I[2] # left bry
    if i==g.nx-1: I[2] = I[1] # right bry
    if j==0:      I[3] = I[4] # bottom bry
    if j==g.ny-1: I[4] = I[3] # top bry
    print I
```

Finite difference approx. of Poisson's eq in 2D (cont.)

As we've seen with the 1D Poisson problem, the system of equations can be represented with a matrix notation. For example, with $k_x = k_y = 1$ and a (3×3) grid ($N = 9$ nodes):

$$\begin{bmatrix} 4 & -2 & & -2 & & & & & \\ -1 & 4 & -1 & & -2 & & & & \\ & -2 & 4 & & & -2 & & & \\ -1 & & & 4 & -2 & & -1 & & \\ & -1 & & -1 & 4 & -1 & & -1 & \\ & & -1 & & -2 & 4 & & & -1 \\ & & & -2 & & & 4 & -2 & \\ & & & & -2 & & -1 & 4 & -1 \\ & & & & & -2 & & -2 & 4 \end{bmatrix} \begin{Bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ u_8 \end{Bmatrix} = \begin{Bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \\ s_7 \\ s_8 \end{Bmatrix}$$

where we accounted for the insulated boundary conditions (Neumann). Note that $\boldsymbol{K}$ is a $(3 \times 3) \times (3 \times 3)$ matrix $(N \times N)$.

Or:

$$\boldsymbol{KU} = \boldsymbol{F}$$

Finite difference approx. of Poisson's eq in 2D (cont.)

Again, the easiest way to account for the essential boundary conditions (Dirichlet or prescribed essential ϕ values) is to split the system into 4 sub-systems:

$$\begin{bmatrix} \boldsymbol{K}_{11} & \boldsymbol{K}_{12} \\ \boldsymbol{K}_{21} & \boldsymbol{K}_{22} \end{bmatrix} \begin{Bmatrix} \boldsymbol{U}_1 \\ \boldsymbol{U}_2 \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{F}_1 \\ \boldsymbol{F}_2 \end{Bmatrix}$$

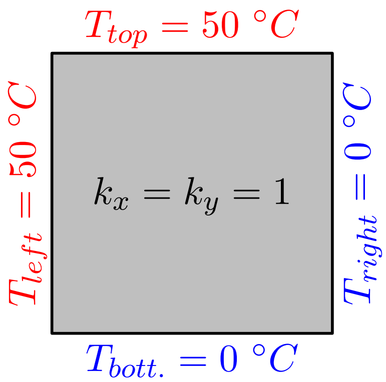
where $\boldsymbol{U}_2$ corresponds to the known/prescribed/essential values.

Therefore, the solution for all other nodes can be found by means of:

$$\boxed{\boldsymbol{U}_1 = \boldsymbol{K}_{11}{}^{-1}(\boldsymbol{F}_1 - \boldsymbol{K}_{12}\boldsymbol{U}_2)}$$

Apart from having to deal with a 2D grid, there is not much difference between the 2D and 1D solution of the Poisson equation via finite differences. The following slides present a **suggestion** of Python implementation. Students are required to develop their own code (can be based on this one, of course).

Example: We want to find out the steady state temperature distribution over a 5x4 plate where its left and top edges are set with a temperature of $50\ ^\circ C$ while its lower and right most edges are set at $0\ ^\circ C$. The conductivities are $k_x = k_y = 1$.

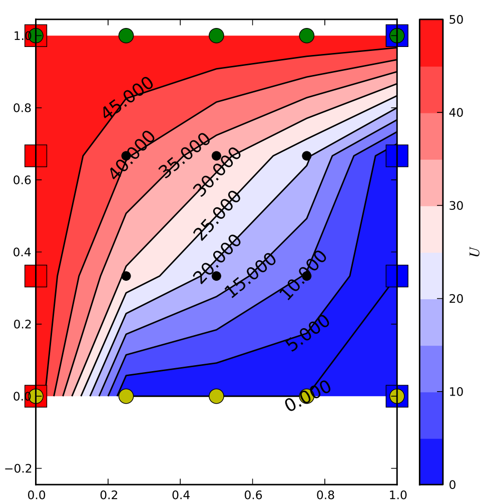


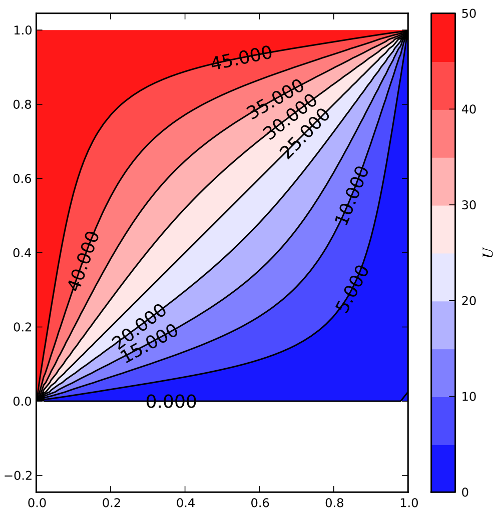

```
from msys_fig import *                  # my routines
from grid2d   import *                  # my 2d grid
def s(x,y): return 0.0                  # source funct
g        = Grid2D(1.,1.,5,4)            # 5x4 grid
kx,  ky  = 1., 1.                       # conductivit.
dxx, dyy = g.dx**2., g.dy**2.           # sq increments
alp      = 2.*(kx/dxx+ky/dyy)           # alpha
bet, gam = -kx/dxx, -ky/dyy             # beta, gamma
mol = [alp, bet, bet, gam, gam]         # molecule
N   = g.nx * g.ny                       # num of nodes
K   = zeros((N,N))                      # K matrix
F   = zeros(N)                          # RHS array
for n in range(N):                      # for each eq
    i, j = n%g.nx, n/g.nx               # col and rows
    I = [n, n-1, n+1, n-g.nx, n+g.nx]   # nodes
    if i==0:        I[1] = I[2]         # left bry
    if i==g.nx-1:   I[2] = I[1]         # right bry
    if j==0:        I[3] = I[4]         # bottom bry
    if j==g.ny-1:   I[4] = I[3]         # top bry
    for p, k in enumerate(I):           # each contrib
        K[n,k] += mol[p]                # set K matrix
    F[n] = s(g.X[i],g.Y[j])             # RHS & source
pn = hstack([g.L, g.R, g.B, g.T])       # presc nods
```

(continued)

The results are presented below for a coarse 5x4 grid and a finer 51x41 grid.

```
U      = zeros(N)           # LHS array
U[g.L] = 50.0; U[g.R]=0.   # presc l,r vals
U[g.T] = 50.0; U[g.B]=0.   # presc t,b vals
eqs    = arange(N)          # all equations
eq2    = eqs[pn]            # prescribed eqs
eq1    = delete(eqs,eq2)    # eqs to be solved
K1_    = K  [eq1,:] # rows 1 of K, any column
K11    = K1_[:,eq1] # any row, cols 1 of K1_
K12    = K1_[:,eq2] # any row, cols 2 of K1_
U[eq1] = solve(K11, F[eq1] - dot(K12, U[eq2]))
```





FDM Conclusions

Finite differences method: summary

Advantages:

- ☐ Is simple and easy to be applied to any system of differential equations
- ☐ Leads to relatively straightforward analyses of accuracy and stability
- ☐ Is simple to be implemented, making it easier to employ techniques for maximum speed (such as parallel computing)

Disadvantages:

- ☐ Has to be modelled for each specific problem
- ☐ Is not designed for non uniform geometries, i.e., works only with rectangular or cubic grids; although some extensions exist
- ☐ Different assumptions must be made to represent natural boundary conditions for different problems

Numerical Methods in Engineering – Part III

Dr Dorival Pedroso

February 27, 2018

Fundamentals

Integration by parts

Given two functions $\alpha = \alpha(x)$ and $\beta = \beta(x)$:

$$\frac{\mathrm{d}\alpha\,\beta}{\mathrm{d}x} = \frac{\mathrm{d}\alpha}{\mathrm{d}x}\,\beta + \alpha\,\frac{\mathrm{d}\beta}{\mathrm{d}x}$$

$$\int_{x_a}^{x_b} \frac{\mathrm{d}\alpha\,\beta}{\mathrm{d}x}\,\mathrm{d}x = \int_{x_a}^{x_b} \frac{\mathrm{d}\alpha}{\mathrm{d}x}\,\beta\,\mathrm{d}x + \int_{x_a}^{x_b} \alpha\,\frac{\mathrm{d}\beta}{\mathrm{d}x}\,\mathrm{d}x$$

$$[\alpha\,\beta]_{x_a}^{x_b} = \int_{x_a}^{x_b} \frac{\mathrm{d}\alpha}{\mathrm{d}x}\,\beta\,\mathrm{d}x + \int_{x_a}^{x_b} \alpha\,\frac{\mathrm{d}\beta}{\mathrm{d}x}\,\mathrm{d}x$$

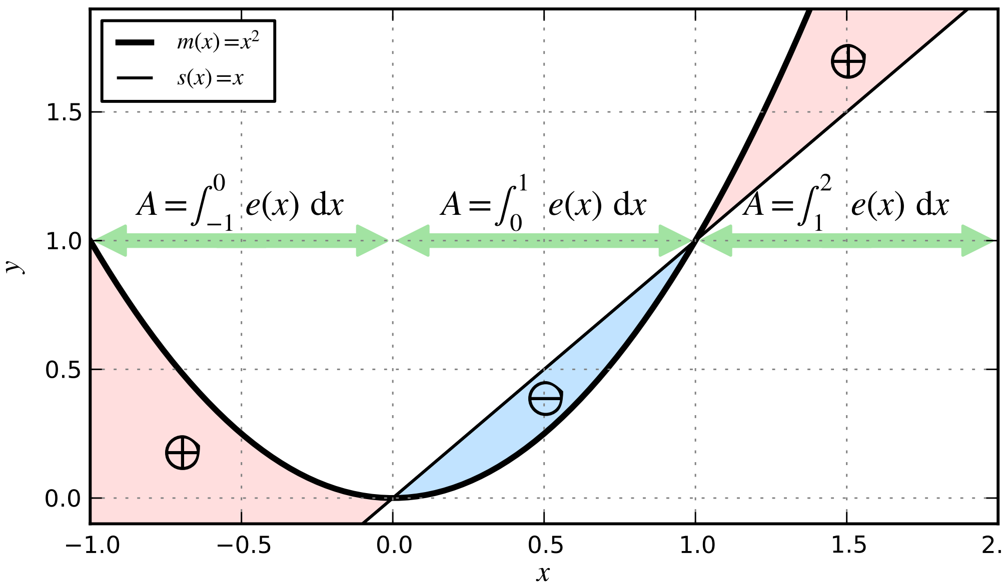
$$\boxed{\int_{x_a}^{x_b} \frac{\mathrm{d}\alpha}{\mathrm{d}x}\,\beta\,\mathrm{d}x = [\alpha\,\beta]_{x_a}^{x_b} - \int_{x_a}^{x_b} \alpha\,\frac{\mathrm{d}\beta}{\mathrm{d}x}\,\mathrm{d}x}$$

Or:

$$\int_{x_a}^{x_b} \frac{\mathrm{d}\alpha}{\mathrm{d}x}\,\beta\,\mathrm{d}x = \alpha(x_b)\,\beta(x_b) - \alpha(x_a)\,\beta(x_a) - \int_{x_a}^{x_b} \alpha\,\frac{\mathrm{d}\beta}{\mathrm{d}x}\,\mathrm{d}x$$

Weighted-Residuals Methods

Suppose we have two curves $m(x)$ and $s(x)$. If these curves are equal to each other, then for every x in the domain of interest, the difference $m(x) - s(x)$ is zero. Otherwise, let's define an **error function** $e(x) = m(x) - s(x)$ and then integrate over a **domain of study** Ω.



If we wish to define a measure of error, or the positive distance between $m(x)$ and $s(x)$, we cannot use $\int e(x)\,\mathrm{d}x$ because this integral (the area between the two curves) sometimes is positive but sometimes negative. Thus, if we add all these parts (integrating), we may get a zero error value, even though the distance between the two curves is not zero!

Another way to define a better error measure, is to add (integrate) the square of the differences in the domain of study. Then, let's define a **total residual** $\bar{R}$ as follows:

$$\bar{R} = \int_{\Omega} e(x)^2\,\mathrm{d}x$$

where Ω representets our domain $[x_0, x_1]$.

Note that $\bar{R}$ is always positive and goes to zero when $e(x)$ is zero. I.e., the residual goes to zero when the two functions $m(x)$ and $s(x)$ "approach" each other. This residual is now a better measure of "correctness", instead of considering $e(x)$ only.

Weighted residuals applied to PDEs

Suppose now that $m(x)$ and $s(x)$ correspond to the following differential equation (1D Poisson):

$$\underbrace{-k_x \frac{\partial^2 u}{\partial x^2}}_{m(x)} = s(x) \quad \text{in} \quad \Omega = \{x \in \mathbb{R} \,|\, 0 \leq x \leq L\}$$

Remember that in this problem we're looking for $u(x)$ for all points in Ω. Thus, if we find the correct $u(x)$, the error $e(x) = m(x) - s(x)$ will also be zero **everywhere** in Ω. And so will be the (total) residual $\bar{R}$.

Alternatively, we may wish to just **approximate** $u(x)$. In this case, the error $e(x)$ may not be zero everywhere (i.e. it may be non-zero somewhere). The residual $\bar{R}$ may assume some (small) positive value then. The better we approximate $u(x)$, the smaller the residual is.

For simple problems, one strategy is to first assume a general expression for $u(x)$ and then **minimise** the residual $\bar{R}$ as much as we can and, **at the same time**, satisfy the **boundary conditions**.

Weighted residuals applied to PDEs (cont.)

Note that, in this strategy, we solve an **averaged integral expression**, instead of directly approximating the derivatives as we did in the finite difference. Also, note that *guessing* a general solution for $u(x)$ is not always easy.

Let's suppose our guess for $u_{correct}(x)$ is an expression $u(x)$ with N coefficients a_i. Example:

$$u(x) = a_0 + a_1\, x + a_2\, x^2 + a_3\, x^3 + a_4\, x^4 \dots$$

Then, to minimise the residual, we have to: (1) differentiate the total residual with respect to a_i; (2) make the result equal to zero; and (3) solve for every a_i (typical minimisation procedure in Calculus). Thus, we have to solve:

$$\frac{\partial \bar{R}}{\partial a_i} = 2 \int_\Omega e(x)\, \frac{\partial e(x)}{\partial a_i}\, \mathrm{d}x = 0$$

or:
$$R_i = \int_\Omega e(x) \frac{\partial e(x)}{\partial a_i}\, \mathrm{d}x = 0$$

Weighted residuals applied to PDEs (cont.)

Therefore, we have to solve N equations, one for each coefficient a_i. Note that each equation contains an integral of the following type:

$$\int_{\Omega} e(x) \underbrace{\frac{\partial e(x)}{\partial a_i}}_{w_i(x)} \, \mathrm{d}x = 0 \quad \text{or:} \quad \boxed{R_i = \int_{\Omega} e(x)\, w_i(x) \, \mathrm{d}x = 0}$$

The N functions $w_i(x)$ are usually known as **weights** and later we'll show that these weights do not need to be equal to $\frac{\partial e(x)}{\partial a_i}$; some other expression that helps with the problem of summing "positive" and "negative" quantities can be adopted instead.

Here, we used the **least-squares** approach to obtain $w_i(x)$, i.e., they are actually the derivative of the error function with respect to the coefficients a_i. Later on, other methods also classified as **weighted-residuals** will be presented.

The strategy of a *weighted-residuals method* is, instead of solving $e(x) = 0$, solve $R_i = \int e(x)\, w_i(x) \mathrm{d}x = 0$ instead in order to get an **approximated** or **weak** solution for $u(x)$.

Weighted residuals applied to PDEs (cont.)

To illustrate, let's apply this strategy to Poisson's equation with $k_x = 1$ and $s(x) = x$. In addition, let's choose for boundary conditions: $u(x=0) = u_0 = 0$ and $u(x=L) = u_L = 0$. Thus, our problem is:

$$-\frac{\partial^2 u}{\partial x^2} = x \quad \text{with} \quad u_0 = 0,\, u_L = 0 \quad \text{in} \quad 0 \leq x \leq L$$

Let's assume initially that our solution can be expressed by:

$$u(x) = a_0 + a_1\, x + a_2\, x^2 + a_3\, x^3 + a_4\, x^4$$

First, let's check that this solution satisfies the boundary conditions:

$$\begin{aligned} u(0) &= a_0 && \neq 0 \\ u(L) &= a_0 + a_1\, L + a_2\, L^2 + a_3\, L^3 + a_4\, L^4 && \neq 0 \end{aligned}$$

I.e., it does not satisfy the boundary conditions – we have to choose another expression then. Let's **try** instead:

$$u(x) = a_0 \sin\left(\frac{\pi\, x}{L}\right)$$

Weighted residuals applied to PDEs (cont.)

Fundamentals
Weighted-Residuals
W-Residuals 1
W-Residuals 2
WR: PDEs 1
WR: PDEs 2
WR: PDEs 3
WR: PDEs 4
▷ WR: PDEs 5
WR: PDEs 6
WR: PDEs 7
WR: PDEs 8
WR: Summary 1
WR: Summary 2
FEM: Introduction
FEM: Formulation
FEM: Implementation
FEM: Examples 1D

This one indeed satisfies the boundary conditions. Now, we can compute the error function $e(x)$:

$$e(x) = -\frac{\partial^2 u}{\partial x^2} - x = \frac{a_0\,\pi^2}{L^2}\,\sin\left(\frac{\pi\,x}{L}\right) - x$$

The weights $w_i(x)$

$$w_0(x) = \frac{\partial e(x)}{\partial a_0} = \frac{\pi^2}{L^2}\sin\left(\frac{\pi\,x}{L}\right)$$

And the weighted-residuals:

$$\int_\Omega e(x)\,w_0(x)\,\mathrm{d}x = \int_\Omega \left[\frac{a_0\,\pi^2}{L^2}\,\sin\left(\frac{\pi\,x}{L}\right) - x\right]\left[\frac{\pi^2}{L^2}\sin\left(\frac{\pi\,x}{L}\right)\right]\,\mathrm{d}x = 0$$

After integration in the domain $0 \le x \le L$:

$$\int_0^L e(x)\,w_0(x) = \frac{\pi^4\,a_0}{2\,L^3} - \pi = 0$$

We get: $a_0 = \frac{2\,L^3}{\pi^3}$

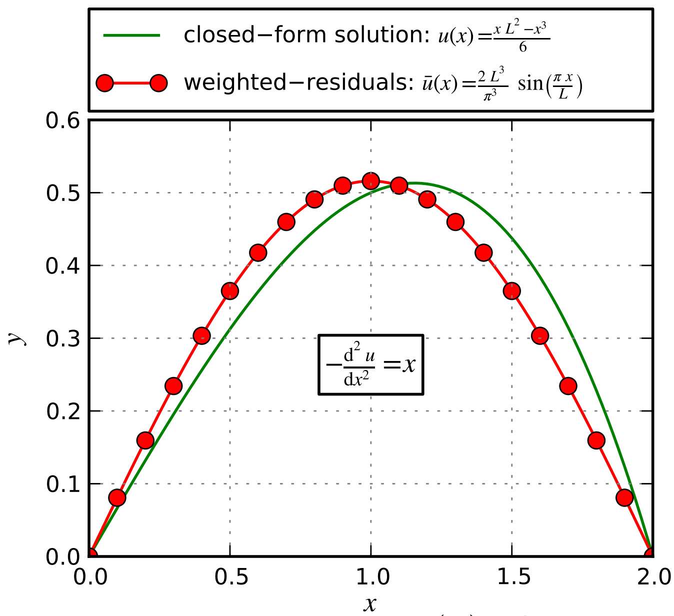
Weighted residuals applied to PDEs (cont.)

Fundamentals
Weighted-Residuals
W-Residuals 1
W-Residuals 2
WR: PDEs 1
WR: PDEs 2
WR: PDEs 3
WR: PDEs 4
WR: PDEs 5
▷ WR: PDEs 6
WR: PDEs 7
WR: PDEs 8
WR: Summary 1
WR: Summary 2
FEM: Introduction
FEM: Formulation
FEM: Implementation
FEM: Examples 1D

Therefore, our approximated solution is (illustrated to the right):

$$u(x) = \frac{2\,L^3}{\pi^3}\,\sin\left(\frac{\pi\,x}{L}\right)$$

The solution is not valid for all points in $0 \le x \le L$.



Note that we've used only one coefficient a_0 to define $u(x)$. Of course we can use more and then solve a **system of equations** for all a_i. So, let's do it again with another guess:

$$u(x) = b\,x + a_0\,x^2 + a_1\,x^3$$

Weighted residuals applied to PDEs (cont.)

To satisfy the boundary conditions:

$$b\,L + a_0\,L^2 + a_1\,L^3 = 0 \quad \Rightarrow \quad b = -a_0\,L - a_1\,L^2$$

Note that $u(0) = 0$ is already satisfied. Then:

$$u(x) = a_0\,x^2 + a_1\,x^3 - (a_0\,L + a_1\,L^2)\,x$$

Computing now the error function, the weights, and the weighted-residuals:

$$e(x) = -\frac{\partial^2 u}{\partial x^2} - x = -2\,a_0 - 6\,a_1\,x - x$$

$$w_0(x) = \frac{\mathrm{d}e(x)}{\mathrm{d}a_0} = -2 \qquad w_1(x) = \frac{\mathrm{d}e(x)}{\mathrm{d}a_1} = -6\,x$$

$$\int_0^L e(x)\,w_0(x)\,\mathrm{d}x = 4\,a_0\,L + 6\,a_1\,L^2 + L^2$$

$$\int_0^L e(x)\,w_1(x)\,\mathrm{d}x = 6\,a_0\,L^2 + 12\,a_1\,L^3 + 2\,L^3$$

Weighted residuals applied to PDEs (cont.)

Therefore, solving the following system for a_0 and a_1:

$$\begin{cases} 4\,a_0\,L + 6\,a_1\,L^2 + L^2 & = 0 \\ 6\,a_0\,L^2 + 12\,a_1\,L^3 + 2\,L^3 & = 0 \end{cases}$$

We get: $a_0 = 0$ and $a_1 = -\frac{1}{6}$ Thus, our weighted-residuals solution is:

$$u(x) = \frac{x\,L^2 - x^3}{6} \quad \text{i.e.} \quad \frac{\partial^2 u(x)}{\partial x^2} = -x \quad \text{and} \quad u_0 = u_L = 0$$

It is interesting to note that the solution above is actually the correct (closed-form) solution! In fact, the weighted-residuals method can be used to obtain exact polynomial solutions, when the guessed expression is able to represent all terms in the correct polynomial.

Of course we don't know in advance the order of the correct solution. Hence, a sort of **trial and error** procedure has to be considered. Nonetheless, in the situation of having to deal with a complex non-linear problem, we can use a **linear** problem instead to start with.

Weighted residuals applied to PDEs: Summary

The weighted-residuals equation $\int_\Omega e(x)\, w_i(x)\, \mathrm{d}x$ is an alternative way of solving $-k_x \frac{\partial^2 u}{\partial x^2} = s(x)$ (for instance). It is known as the **weak form**, since the integral expression makes the **total** (summed) error go to zero but does not necessarily satisfy the differential equation for **all** x values.

On the other hand, the original differential equation is known as the **strong form** because it requires the error $e(x)$ to vanish at every x.

Other weighted-residuals methods can be developed by choosing different expressions for the weights $w_i(x)$

For example, a popular method is the **Galerkin method** (after Boris Grigoryevich Galerkin, 1871-1945), where the weights are defined as the derivatives of the **trial (guessed/assumed)** solution: $w_i(x) = \frac{\partial u(x)}{\partial a_i}$.

Note that the method we have used before is the **least-squares** method where the weights are the derivatives of the **error** function: $w_i(x) = \frac{\partial e(x)}{\partial a_i}$.

Weighted residuals applied to PDEs: Summary (cont.)

Fundamentals

Weighted-Residuals
W-Residuals 1
W-Residuals 2
WR: PDEs 1
WR: PDEs 2
WR: PDEs 3
WR: PDEs 4
WR: PDEs 5
WR: PDEs 6
WR: PDEs 7
WR: PDEs 8
WR: Summary 1
▷ WR: Summary 2

FEM: Introduction

FEM: Formulation

FEM: Implementation

FEM: Examples 1D

Other weighted-residuals methods can use arbitrarily chosen weights or weights based on the Dirac δ function. The first is known as Petrov-Galerkin and the latter as collocation.

Table 1: Common weighted-residuals methods

Method	Weights	Comments
Least-squares	$w_k = \frac{\partial e}{\partial a_k}$	weights are the derivatives of the error function
(Bubnov-)Galerkin	$w_k = \frac{\partial u}{\partial a_k}$	weights are the derivatives of the assumed solution
Petrov-Galerkin	$w_k = \psi(x)$	weights are chosen arbitrarily
Collocation	$w_k = \delta(x - x_k)$	weights depend on selected x_k points ($\delta(x)$ is the Dirac function)

Introduction of the Finite Element Method (FEM)

Introduction to the Finite Element Method (FEM)

The main drawback of weighted-residuals methods is that there are no guidelines for an appropriate choice of **admissible** trial solutions. Admissible means that we have to satisfy also the boundary conditions!

Based on the previous examples, it can be seen that this trial-and-error procedure may become daunting, especially for more complex equations.

To overcome these problems, the **finite element** method splits the weighted-residuals integral into a **finite** number of smaller parts that can be represented by simpler functions. These parts can then be added (**assembled**) later.

Intro to the Finite Element Method (FEM) (cont.)

Fundamentals
Weighted-Residuals
FEM: Introduction
Introduction 1
▷ Introduction 2
Introduction 3
Introduction 4
Introduction 5
Introduction 6
Introduction 7
Introduction 8
Introduction 9
Introduction 10
Introduction 11
Introduction 12
Introduction 13
Introduction 14
FEM: Formulation
FEM: Implementation
FEM: Examples 1D

In the FEM, the integral form (weak form) is evaluated separately over each smaller sub-domain (**element**) and then added as follows:

$$\int_{x_0}^{x_1} e(x)\, w_i(x)\,\mathrm{d}x = \int_{x_0}^{x_a} e_a(x)\, w_{ia}(x)\,\mathrm{d}x + \int_{x_a}^{x_b} e_b(x)\, w_{ib}(x)\,\mathrm{d}x + \ldots + \int_{x_b}^{x_1} e_n(x)\, w_{in}(x)\,\mathrm{d}x$$

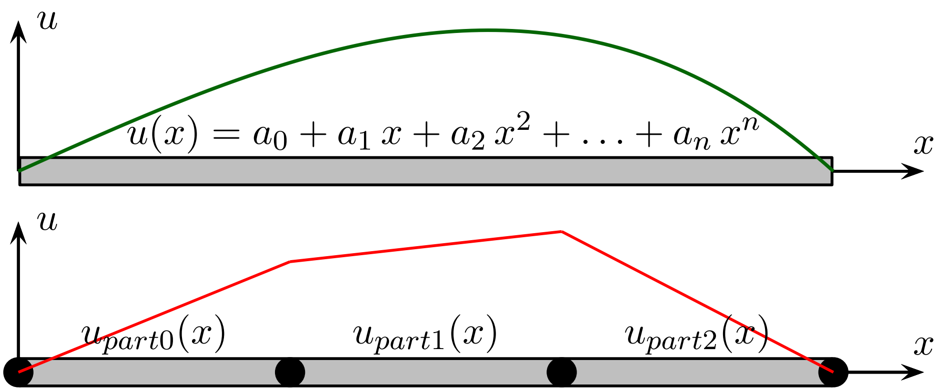
Also, the trial expressions can now be developed in such a way to **automatically** satisfy the boundary conditions at the "extremes" of the element, i.e. at the element **nodes**.

After the **assemblage** of all elements, the boundary conditions will then be satisfied elsewhere in Ω.

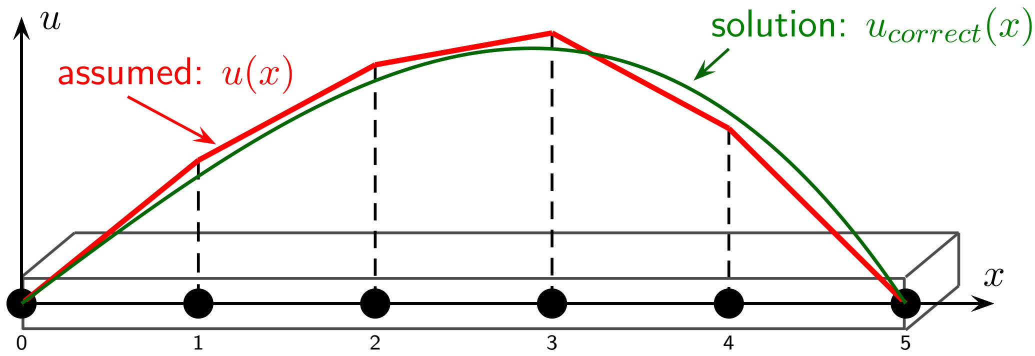
Intro to the Finite Element Method (FEM) (cont.)

Fundamentals
Weighted-Residuals
FEM: Introduction
Introduction 1
Introduction 2
▷ Introduction 3
Introduction 4
Introduction 5
Introduction 6
Introduction 7
Introduction 8
Introduction 9
Introduction 10
Introduction 11
Introduction 12
Introduction 13
Introduction 14
FEM: Formulation
FEM: Implementation
FEM: Examples 1D

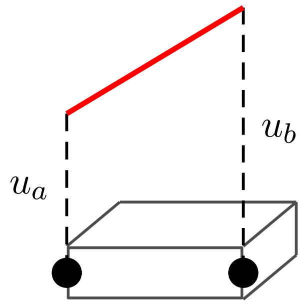
Now, **within a single element**, simple expressions can be assumed for the general solution $u(x)$. They do not necessarily require continuum first order derivatives $\frac{\partial u}{\partial x}$ from element to element. This simplifies a lot the choice of expressions.



For example, consider the following one-dimensional problem, where the solution domain is divided into five parts:



Each part can be easily represented as follows:



$$u_e(x) = \frac{x_b - x}{x_b - x_a}\, u_a + \frac{x - x_a}{x_b - x_a}\, u_b$$

The good thing now is that the (essential) boundary conditions can be easily set. For instance, for the left-most element in the figure (setting: $x_a = 0$, $x_b = 1$, $u_a = 0$, $u_b = u_1$):

$$u_{e0}(x) = x\, u_1 \qquad 0 \leq x \leq 1$$

And for the right-most element (setting: $x_a = 4$, $x_b = 5$, $u_a = u_4$, $u_b = 0$):

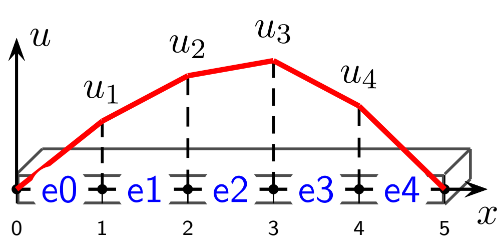
$$u_{e4}(x) = (5 - x)\, u_4 \qquad 4 \leq x \leq 5$$

Example problem with:

$u(x = 0) = 0$ and $u(x = L) = 0$

Assumption:

$$u_e(x) = \frac{x_b - x}{x_b - x_a}\, u_a + \frac{x - x_a}{x_b - x_a}\, u_b$$



Both satisfying all boundary conditions. Now we just need to find u_1, u_2 and u_3 in order to **minimise the residuals**.

Note that the coefficients to be determined ($u_i = a_i$) have now a better meaning: they are the **nodal** values of the **FEM mesh**!

Intro to the Finite Element Method (FEM) (cont.)

Fundamentals
Weighted-Residuals
FEM: Introduction
Introduction 1
Introduction 2
Introduction 3
Introduction 4
Introduction 5
▷ Introduction 6
Introduction 7
Introduction 8
Introduction 9
Introduction 10
Introduction 11
Introduction 12
Introduction 13
Introduction 14
FEM: Formulation
FEM: Implementation
FEM: Examples 1D

To illustrate, let's complete the previous example. The other assumed equations are:

$$
\begin{aligned}
u_{e1}(x) &= (2-x)\,u_1 + (x-1)\,u_2 \qquad \text{for} \quad 1 \le x \le 2\\
u_{e2}(x) &= (3-x)\,u_2 + (x-2)\,u_3 \qquad \text{for} \quad 2 \le x \le 3\\
u_{e3}(x) &= (4-x)\,u_3 + (x-3)\,u_4 \qquad \text{for} \quad 3 \le x \le 4
\end{aligned}
$$

Note that in order to calculate $\frac{\partial^2 u}{\partial x^2}$ and then the error functions $e_{ei}(x)$, the assumed/guessed general expressions should have at least a quadratic term, otherwise all second order derivatives would be zero. Implying zero residuals automatically!

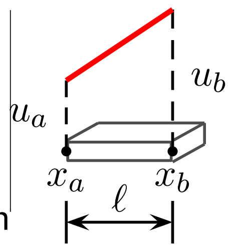
This is the case with our previous **linear piecewise discretisation**. A way to solve this now and still be able to use simple linear expressions is to **lower** the order of $\frac{\partial^2 u}{\partial x^2}$ in the weighted-residuals expressions.

This can be done by means of the **Green-Gauss divergence** theorem (**integration by parts in 1D**).

Intro to the Finite Element Method (FEM) (cont.)

Fundamentals
Weighted-Residuals
FEM: Introduction
Introduction 1
Introduction 2
Introduction 3
Introduction 4
Introduction 5
Introduction 6
▷ Introduction 7
Introduction 8
Introduction 9
Introduction 10
Introduction 11
Introduction 12
Introduction 13
Introduction 14
FEM: Formulation
FEM: Implementation
FEM: Examples 1D

Let's work now with a **typical element** $\Omega_e = \{x \in \mathbb{R} \,|\, x_a \le x \le x_b\}$ and let's use $u(x)$ to indicate $u_e(x)$ in this domain. Note that we can always substitute later the conditions $u_0 = 0$ and $u_L = 0$ at the left- and right-most elements. Our assumed solution from x_a to x_b is:



$$
u(x) = \color{red}{u_e(x)} = \frac{x_b - x}{\ell}\,u_a + \frac{x - x_a}{\ell}\,u_b
$$

Considering an auxiliary variable $\bar{v}_x$ (seepage/diffusion velocity):

$$
\bar{v}_x = -\frac{\partial u}{\partial x} \qquad \text{thus:} \qquad \frac{\partial \bar{v}_x}{\partial x} = -\frac{\partial^2 u}{\partial x^2}
$$

Then, for the simple problem: $-\frac{\partial^2 u}{\partial x^2} = 0$, the error function will be:

$$
e(x) = -\frac{\partial^2 u}{\partial x^2} - 0 = \frac{\partial \bar{v}_x}{\partial x}
$$

with corresponding residuals $R_i = \int_\Omega e(x)\,w_i(x)\,\mathrm{d}x = 0$.

For a typical element Ω^e and $e(x) = -\frac{\partial^2 u}{\partial x^2} = \frac{\partial \bar{v}_x}{\partial x}$, the weighted-residuals are:

$$R_i = \int_{\Omega^e} e(x)\, w_i(x)\, \mathrm{d}x = \int_{\Omega^e} \frac{\partial \bar{v}_x}{\partial x}\, w_i\, \mathrm{d}x = 0$$

with $w_i = w_i(x)$.

Applying integration by parts:

$$\int_{\Omega^e} \frac{\partial \bar{v}_x}{\partial x}\, w_i\, \mathrm{d}x = \left[\bar{v}_x\, w_i\right]_{x_a}^{x_b} - \int_{\Omega^e} \bar{v}_x\, \frac{\partial w_i}{\partial x}\, \mathrm{d}x$$

Then, after substituting $\bar{v}_x = -\frac{\partial u}{\partial x}$ back:

$$\boxed{\int_{\Omega^e} e(x)\, w_i(x)\, \mathrm{d}x = \int_{\Omega^e} \frac{\partial u}{\partial x}\, \frac{\partial w_i}{\partial x}\, \mathrm{d}x - \left[\frac{\partial u}{\partial x}\, w_i\right]_{x_a}^{x_b}}$$

Note that the term $\frac{\partial^2 u}{\partial x^2}$ disappeared from the weighted-residuals expression. Indeed, one derivative "has moved" from u to w_i and hence both derivatives are of first order.

Note also that the term $\left[\frac{\partial u}{\partial x}\, w_i\right]_{x_a}^{x_b}$ is only non-zero at the boundaries and depends on the boundary conditions. In this example, we have $\frac{\partial u}{\partial x}\Big|_{boundaries} = 0$; thus this term is zero.

To continue, instead of using the least-squares method to compute w_i, we will employ the **Galerkin** method, where the weights are the derivatives of the assumed solution with respect to its own coefficients:

$$w_i(x) = \frac{\partial u(x)}{\partial u_i}$$

Intro to the Finite Element Method (FEM) (cont.)

Fundamentals

Weighted-Residuals

FEM: Introduction
Introduction 1
Introduction 2
Introduction 3
Introduction 4
Introduction 5
Introduction 6
Introduction 7
Introduction 8
Introduction 9
▷ Introduction 10
Introduction 11
Introduction 12
Introduction 13
Introduction 14

FEM: Formulation

FEM: Implementation

FEM: Examples 1D

Employing the Galerking method:

$$w_a = \frac{\partial u(x)}{\partial u_a} = \frac{x_b - x}{\ell} = S_a(x)$$

and:

$$w_b = \frac{\partial u(x)}{\partial u_b} = \frac{x - x_a}{\ell} = S_b(x)$$

The derivatives of w_i are:

$$\frac{\partial w_a}{\partial x} = \frac{-1}{\ell} = G_a(x)$$

and:

$$\frac{\partial w_b}{\partial x} = \frac{1}{\ell} = G_b(x)$$

Also, note that:

$$\frac{\partial u}{\partial x} = \frac{-1}{\ell} u_a + \frac{1}{\ell} u_b = G_a\, u_a + G_b\, u_b$$

Intro to the Finite Element Method (FEM) (cont.)

Fundamentals

Weighted-Residuals

FEM: Introduction
Introduction 1
Introduction 2
Introduction 3
Introduction 4
Introduction 5
Introduction 6
Introduction 7
Introduction 8
Introduction 9
Introduction 10
▷ Introduction 11
Introduction 12
Introduction 13
Introduction 14

FEM: Formulation

FEM: Implementation

FEM: Examples 1D

Then, the weighted-residuals expressions can be computed as follows:

$$\int_{\Omega^e} \frac{\partial u}{\partial x}\frac{\partial w_a}{\partial x}\,\mathrm{d}x = \int_{x_a}^{x_b} \left(\frac{-1}{\ell}u_a + \frac{1}{\ell}u_b\right)\left(\frac{-1}{\ell}\right)\mathrm{d}x = \frac{1}{\ell}u_a - \frac{1}{\ell}u_b$$

$$\int_{\Omega^e} \frac{\partial u}{\partial x}\frac{\partial w_b}{\partial x}\,\mathrm{d}x = \int_{x_a}^{x_b} \left(\frac{-1}{\ell}u_a + \frac{1}{\ell}u_b\right)\left(\frac{1}{\ell}\right)\mathrm{d}x = \frac{-1}{\ell}u_a + \frac{1}{\ell}u_b$$

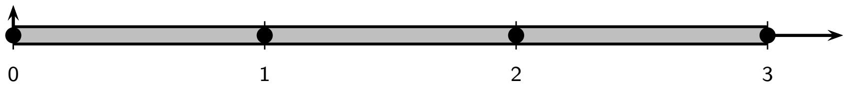
With a **mesh** of N nodes, a linear system with N equations is obtained. Each element will **add** (contribute) to two equations. For example, with 3 elements and 4 nodes:

$$\begin{array}{lllllll}
\int_0^3 e(x)\,w_0(x) = & \int_0^1 e\,w_0\mathrm{d}x & & & & & = 0 \\
\int_0^3 e(x)\,w_1(x) = & \int_0^1 e\,w_1\mathrm{d}x & + & \int_1^2 e\,w_1\mathrm{d}x & & & = 0 \\
\int_0^3 e(x)\,w_2(x) = & & & \int_1^2 e\,w_2\mathrm{d}x & + & \int_2^3 e\,w_2\mathrm{d}x & = 0 \\
\int_0^3 e(x)\,w_3(x) = & & & & & \int_2^3 e\,w_3\mathrm{d}x & = 0
\end{array}$$

Intro to the Finite Element Method (FEM) (cont.)

Fundamentals
Weighted-Residuals
FEM: Introduction
Introduction 1
Introduction 2
Introduction 3
Introduction 4
Introduction 5
Introduction 6
Introduction 7
Introduction 8
Introduction 9
Introduction 10
Introduction 11
▷ Introduction 12
Introduction 13
Introduction 14
FEM: Formulation
FEM: Implementation
FEM: Examples 1D

Substituting, observing that $\ell = 1$:



$$\begin{array}{lcccccl}
\int_0^3 e(x)\, w_0(x) = & u_0 - u_1 & & & & & = 0 \\
\int_0^3 e(x)\, w_1(x) = & -u_0 + u_1 & + & u_1 - u_2 & & & = 0 \\
\int_0^3 e(x)\, w_2(x) = & & & -u_1 + u_2 & + & u_2 - u_3 & = 0 \\
\int_0^3 e(x)\, w_3(x) = & & & & & -u_2 + u_3 & = 0
\end{array}$$

Or:

$$\begin{bmatrix} 1 & -1 & & \\ -1 & 2 & -1 & \\ & -1 & 2 & -1 \\ & & -1 & 1 \end{bmatrix} \begin{Bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{Bmatrix}$$

Again, the determinant is zero, which means this system cannot be solved until the boundary conditions are prescribed.

Intro to the Finite Element Method (FEM) (cont.)

Fundamentals
Weighted-Residuals
FEM: Introduction
Introduction 1
Introduction 2
Introduction 3
Introduction 4
Introduction 5
Introduction 6
Introduction 7
Introduction 8
Introduction 9
Introduction 10
Introduction 11
Introduction 12
▷ Introduction 13
Introduction 14
FEM: Formulation
FEM: Implementation
FEM: Examples 1D

To illustrate, let's also consider a source term: $s(x) = x$. Hence:

$$e(x) = -\frac{\partial^2 u}{\partial x^2} - x = \frac{\partial \bar{v}_x}{\partial x} - x \quad \text{and} \quad R_i = \int_{\Omega^e} \left(\frac{\partial \bar{v}_x}{\partial x} - x \right) w_i \, \mathrm{d}x = 0$$

The first term on the integral leads to the same equations as before. The second term becomes:

$$\int_{\Omega^e} -x\, w_a \, \mathrm{d}x = \int_{x_a}^{x_b} -x \, \frac{x_b - x}{\ell} \, \mathrm{d}x = -\frac{\ell}{6}(2\, x_a + x_b)$$

$$\int_{\Omega^e} -x\, w_b \, \mathrm{d}x = \int_{x_a}^{x_b} -x \, \frac{x - x_a}{\ell} \, \mathrm{d}x = -\frac{\ell}{6}(x_a + 2\, x_b)$$

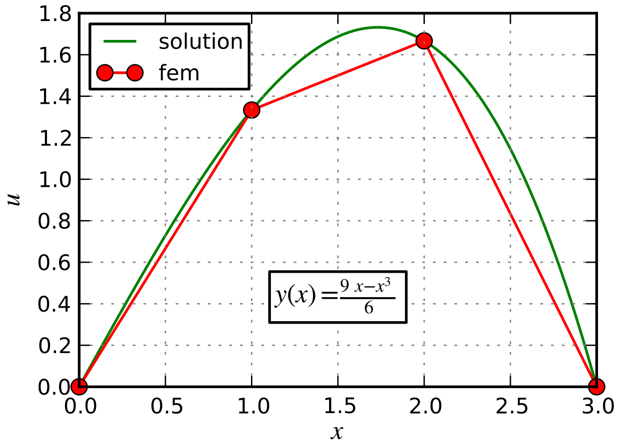
Thus, for a mesh with 4 nodes and $\ell = 1$:

$$\begin{array}{lccc}
\int_0^3 -x\, w_0(x) = & -1/6 & & \\
\int_0^3 -x\, w_1(x) = & -2/6 & -4/6 & \\
\int_0^3 -x\, w_2(x) = & & -5/6 & -7/6 \\
\int_0^3 -x\, w_3(x) = & & & -8/6
\end{array}$$

With $u_0 = 0$ and $u_3 = 0$, we then have to solve the following system:

$$\begin{bmatrix} 1 & -1 & & \\ -1 & 2 & -1 & \\ & -1 & 2 & -1 \\ & & -1 & 1 \end{bmatrix} \begin{Bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{Bmatrix} = \begin{Bmatrix} 1/6 \\ 1 \\ 2 \\ 8/6 \end{Bmatrix}$$

which can only be done after removing the first and last equations.



Introduction to a Matrix Formulation of the Finite Element Method (FEM)

Matrix formulation of the FEM

In order to **automatise** the finite element method, a matrix formulation is developed. It can be seen that the FEM naturally generates a number of matrices and vectors. For instance, for the problem $-\frac{\partial^2 u}{\partial x^2} = x$, each element produces the following **contributions** to the linear system:

$$\int_{\Omega^e} \frac{\partial u}{\partial x}\frac{\partial w_a}{\partial x}\,\mathrm{d}x = \frac{1}{\ell}u_a - \frac{1}{\ell}u_b$$

$$\int_{\Omega^e} \frac{\partial u}{\partial x}\frac{\partial w_b}{\partial x}\,\mathrm{d}x = \frac{-1}{\ell}u_a + \frac{1}{\ell}u_b$$

and

$$\int_{\Omega^e} -x\,w_a\,\mathrm{d}x = -\frac{\ell}{6}(2\,x_a + x_b)$$

$$\int_{\Omega^e} -x\,w_b\,\mathrm{d}x = -\frac{\ell}{6}(x_a + 2\,x_b)$$

or, in matrix form:

$$\begin{Bmatrix} \int_{x_a}^{x_b} e\,w_a\,\mathrm{d}x \\ \int_{x_a}^{x_b} e\,w_b\,\mathrm{d}x \end{Bmatrix} = \frac{1}{\ell}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}\begin{Bmatrix} u_a \\ u_b \end{Bmatrix} - \frac{\ell}{6}\begin{Bmatrix} 2\,x_a + x_b \\ x_a + 2\,x_b \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix}$$

Matrix formulation of the FEM (cont.)

Therefore, for each element, a small matrix $\boldsymbol{K}^e$ and a small vector $\boldsymbol{F}^e$ can be written first, considering the **local node numbers "a" and "b"**, and then later assembled to the larger system. For example:

$$\boldsymbol{K}^e\boldsymbol{U}^e = \boldsymbol{F}^e$$

with:

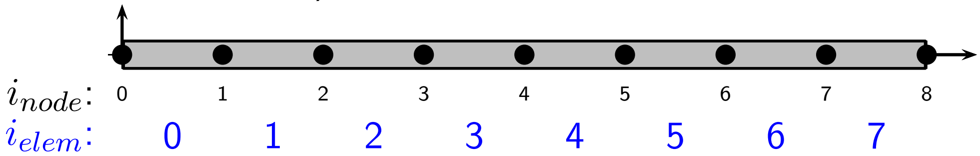
$$\boldsymbol{K}^e = \frac{1}{\ell}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \quad \boldsymbol{U}^e = \begin{Bmatrix} u_a \\ u_b \end{Bmatrix} \quad \boldsymbol{F}^e = \frac{\ell}{6}\begin{Bmatrix} 2\,x_a + x_b \\ x_a + 2\,x_b \end{Bmatrix}$$

Later, the corresponding contributions can be added (**assembled**) to the larger matrix $\boldsymbol{K}$ and vector $\boldsymbol{F}$. For instance, considering the previous example, $\boldsymbol{K}\boldsymbol{U} = \boldsymbol{F}$ is:

$$\begin{bmatrix} 1 & -1 & & \\ -1 & 1+1 & -1 & \\ & -1 & 1+1 & -1 \\ & & -1 & 1 \end{bmatrix}\begin{Bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{Bmatrix} = \begin{Bmatrix} 1/6 \\ 2/6 + 4/6 \\ 5/6 + 7/6 \\ 8/6 \end{Bmatrix}$$

The key for an automatic assembly is the definition of a mapping of **local node numbers** to **global node numbers**. Moreover, each node may contain more than one component of u to be solved. For example, in mechanical problems u_x and u_y can be the horizontal and vertical displacements, respectively, at each node. In this case, these variables, that are well known as **degrees of freedom** in mechanical problems, have to be mapped from local equations to global equations.

In 1D problems with one solution-variable (SOV) u (or degree-of-freedom, DOF) per node, the mapping is quite simple:



$$i_a = i_e, \quad i_b = i_e + 1 \quad \text{and} \quad map = [i_a, i_b]$$

where i_a is the **global** index of the **a** node, i_b is the **global** index of the **b** node, and i_e is the index of the element.

The computer implementation of a 1D FEM code is quite straightforward. For this simple 1D Poisson equation: $-\frac{\partial^2 u}{\partial x^2} = x$, the only difference with respect to the previous FDM code is the way we *assemble* the arrays $\boldsymbol{K}$ and $\boldsymbol{F}$. This is illustrated in the Python script to the right.

The second part of the code (next slide) specifying prescribed boundary conditions and solving the modified (*split*) system of equations does not change at all:

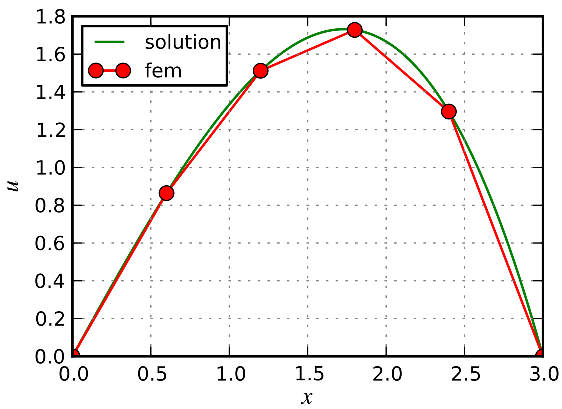
$$\boldsymbol{U_1} = \boldsymbol{K_{11}}^{-1}(\boldsymbol{F_1} - \boldsymbol{K_{12}}\boldsymbol{U_2})$$

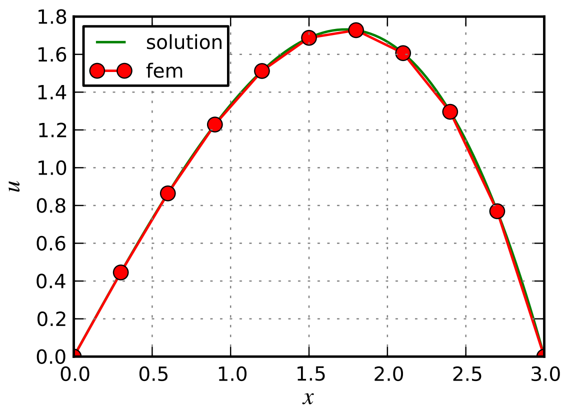
```
from msys_fig import *
# data
L  = 3.0               # length of bar
nn = 6                 # number of nodes
ne = nn-1              # number of elements
l  = L / ne            # length of element
# assemble K and F
K = zeros((nn,nn))     # global matrix K
F = zeros(nn)          # global vector F
for ie in range(ne):   # for each element
    ia, ib = ie,   ie+1 # global indices of nod
    xa, xb = ia*l, ib*l # nodal coordinates
    m = [ia, ib]        # map local to global
    # local (element) matrix Ke
    Ke = (1./l) * array([[ 1.,-1.],
                         [-1., 1.]])
    # local (element) vector Fe
    Fe = (l/6.) * array([[2.*xa+xb],
                         [xa+2.*xb]])
    # assembly
    for i in range(2):      # for each row of Ke
        for j in range(2): # for each col of Ke
            K[m[i],m[j]] += Ke[i,j] # assemble K
        F[m[i]] += Fe[i]    # assemble F
```

(continued)

Note that the script part to the right is pretty much the same as the one used in the FDM. The results after running this script are shown below for 6 nodes (left) and 11 nodes (right).

```
# solve
U      = zeros(nn)        # U vector
pn     = [0, nn-1]        # prescribed nodes
U[pn]  = [0.0, 0.0]       # prescribed vals
eqs    = arange(nn)       # all equations
eq2    = eqs[pn]          # prescribed eqs
eq1    = delete(eqs,eq2)  # eqs to be solved
K1_    = K  [eq1,:] # rows 1 of K, any column
K11    = K1_[:,eq1] # any row, cols 1 of K1_
K12    = K1_[:,eq2] # any row, cols 2 of K1_
U[eq1] = solve(K11, F[eq1] - dot(K12, U[eq2]))
```





Let's derive the FEM formulation for the following (diffusion) problem:

$$\boxed{\rho \frac{\partial u}{\partial t} - k_x \frac{\partial^2 u}{\partial x^2} + \beta\, u = s(x)}$$

using 1D elements with 2 nodes (**"0"** and **"1"**, instead of "a" and "b"). The boundary conditions will be specified later.

The assumed solution, in matrix form, is:

$$u(x) = \begin{bmatrix} \underbrace{\frac{x_1 - x}{\ell}}_{S_0(x)} & \underbrace{\frac{x - x_0}{\ell}}_{S_1(x)} \end{bmatrix} \begin{Bmatrix} u_0 \\ u_1 \end{Bmatrix} \quad \text{or} \quad u(x) = \boldsymbol{S} \boldsymbol{U}^e$$

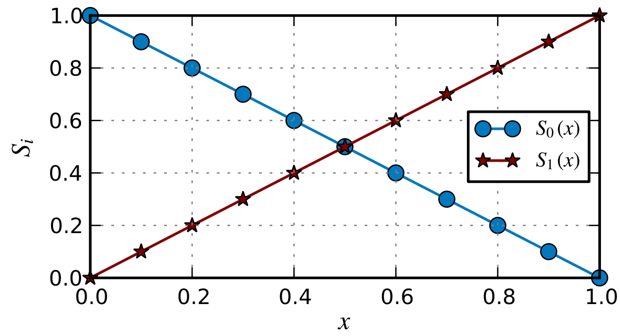
The vector $\boldsymbol{U}^e$ is known as vector of **nodal values** and the functions $S_i(x)$ are known as **shape or interpolation functions**. These functions carry some important properties such as being equal to 1, at the node i, and being equal to zero at any other node of the element.

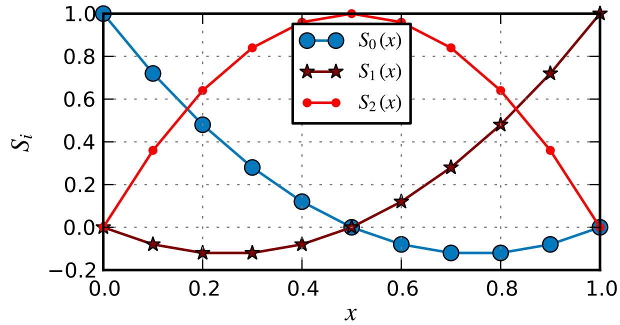
Matrix formulation of the FEM (cont.)

Fundamentals
Weighted-Residuals
FEM: Introduction
FEM: Formulation
Matrix form 1
Matrix form 2
Matrix form 3
Matrix form 4
Matrix form 5
1D Diffusion 1
▷ 1D Diffusion 2
1D Diffusion 3
1D Diffusion 4
1D Diffusion 5
1D Diffusion 6
1D Diffusion 7
1D Diffusion 8
1D Diffusion 9
1D Diffusion 10
1D Diffusion 11
Matrix: summary
FEM: Implementation
FEM: Examples 1D

The two shape functions of a **linear** element are plotted in the figure to the right (with $x_0 = 0$ and $x_1 = 1$).

Elements with **more nodes in the element** are possible as well. These lead to higher order **interpolation** characteristics. For instance, although not used here, the three shape functions of a **quadratic** element are illustrated to the right.





Matrix formulation of the FEM (cont.)

Fundamentals
Weighted-Residuals
FEM: Introduction
FEM: Formulation
Matrix form 1
Matrix form 2
Matrix form 3
Matrix form 4
Matrix form 5
1D Diffusion 1
1D Diffusion 2
▷ 1D Diffusion 3
1D Diffusion 4
1D Diffusion 5
1D Diffusion 6
1D Diffusion 7
1D Diffusion 8
1D Diffusion 9
1D Diffusion 10
1D Diffusion 11
Matrix: summary
FEM: Implementation
FEM: Examples 1D

Considering the Galerkin method, i.e.:

$$w_i(x) = \frac{\partial u}{\partial u_i} = S_i(x)$$

the weighted-residuals expression for one element is:

$$R_i = \int_{\Omega^e} e(x)\, S_i(x)\, \mathrm{d}x = 0$$

or in matrix form, the **two equations** can be written as follows:

$$\int_{\Omega^e} e(x) \begin{Bmatrix} S_0(x) \\ S_1(x) \end{Bmatrix} \mathrm{d}x = 0 \quad \text{or} \quad \int_{\Omega^e} \boldsymbol{S}^T e(x)\, \mathrm{d}x = 0$$

The rate of u can also be written in matrix form:

$$\frac{\partial u}{\partial t} = \dot{u} = \begin{bmatrix} S_0(x) & S_1(x) \end{bmatrix} \begin{Bmatrix} \frac{\partial u_0}{\partial t} \\ \frac{\partial u_1}{\partial t} \end{Bmatrix} = \boldsymbol{S}\dot{\boldsymbol{U}}^e$$

since the shape functions do not change in time (assumed).

Matrix formulation of the FEM (cont.)

Let's introduce again an auxiliary variable $\bar{v}_x$ (the seepage/diffusion velocity); however slightly different this time:

$$\begin{aligned}\bar{v}_x = -k_x \frac{\partial u}{\partial x} &= -k_x \left[\tfrac{\partial}{\partial x}\left(\tfrac{x_1 - x}{\ell}\right) \quad \tfrac{\partial}{\partial x}\left(\tfrac{x - x_0}{\ell}\right)\right] \begin{Bmatrix} u_0 \\ u_1 \end{Bmatrix} \\ &= -k_x \left[\underbrace{\frac{-1}{\ell}}_{G_0(x)} \quad \underbrace{\frac{1}{\ell}}_{G_1(x)} \right] \begin{Bmatrix} u_0 \\ u_1 \end{Bmatrix} \\ &= -k_x \, \boldsymbol{G} \boldsymbol{U}^e \end{aligned}$$

where

$$\boldsymbol{G} = \left[\frac{\partial S_0(x)}{\partial x} \quad \frac{\partial S_1(x)}{\partial x} \right] = \left[G_0(x) \quad G_1(x) \right]$$

is the matrix "**gradient** of shape functions".

Note that:

$$\frac{\partial \bar{v}_x}{\partial x} = -k_x \frac{\partial^2 u}{\partial x^2}$$

for homogeneous k_x (not function of x).

Matrix formulation of the FEM (cont.)

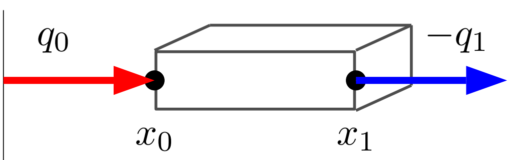
Now, recalling the integration by parts:

$$\int_{\Omega^e} \frac{\partial \bar{v}_x}{\partial x} S_i \, \mathrm{d}x = \left[\bar{v}_x \, S_i \right]_{x_0}^{x_1} - \int_{\Omega^e} \bar{v}_x \frac{\partial S_i}{\partial x} \mathrm{d}x$$

where, the two expressions can be grouped as follows:

$$\int_{\Omega^e} \boldsymbol{S}^T \frac{\partial \bar{v}_x}{\partial x} \mathrm{d}x = \left[\boldsymbol{S}^T \, \bar{v}_x \right]_{x_0}^{x_1} - \int_{\Omega^e} \boldsymbol{G}^T \, \bar{v}_x \, \mathrm{d}x$$

The term $[\bar{v}_x \, S_i]_{x_0}^{x_1}$ depends on the **natural/flux** boundary conditions; however we can choose some **default** values and later replace them as appropriate. Let's assume q_i as positive when substance is flowing **into** the element.



$$\begin{aligned}\left[\bar{v}_x \, S_0 \right]_{x_0}^{x_1} &= \cancel{-q_1 \, S_0(x_1)} - q_0 \, S_0(x_0) \\ &= -q_0\end{aligned}$$

$$\begin{aligned}\left[\bar{v}_x \, S_1 \right]_{x_0}^{x_1} &= -q_1 \, S_1(x_1) - \cancel{q_0 \, S_1(x_0)} \\ &= -q_1\end{aligned}$$

Or, in matrix notation:

$$
\begin{aligned}
\left[\boldsymbol{S}^T\,\bar{v}_x\right]_{x_0}^{x_1} &= \begin{Bmatrix} S_0(x_1) \\ S_1(x_1) \end{Bmatrix} (-q_1) - \begin{Bmatrix} S_0(x_0) \\ S_1(x_0) \end{Bmatrix} q_0 \\
&= \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} (-q_1) - \begin{Bmatrix} 1 \\ 0 \end{Bmatrix} q_0 \\
&= -\begin{Bmatrix} q_0 \\ q_1 \end{Bmatrix}
\end{aligned}
$$

Thus, recalling that $\bar{v}_x = -k_x \boldsymbol{G}\boldsymbol{U}^e$, the integration by parts of the term including the gradient of $\bar{v}_x$ results in:

$$
\boxed{\int_{\Omega^e} \boldsymbol{S}^T \frac{\partial \bar{v}_x}{\partial x}\,\mathrm{d}x = -\begin{Bmatrix} q_0 \\ q_1 \end{Bmatrix} + \int_{\Omega^e} k_x \boldsymbol{G}^T\,\boldsymbol{G}\boldsymbol{U}^e\,\mathrm{d}x}
$$

Now, with the following error function:

$$
e(x) = \rho\,\dot{u} + \beta\,u - s(x) + \frac{\partial \bar{v}_x}{\partial x}
$$

The weighted-residuals expression for **one element** can be developed, after recalling that $\int \boldsymbol{S}^T e(x)\,\mathrm{d}x = 0$ and substituting $u = \boldsymbol{S}\boldsymbol{U}^e$, $\dot{u} = \boldsymbol{S}\dot{\boldsymbol{U}}^e$ and the previous integration-by-parts expression:

$$
\begin{array}{lll}
0 = \displaystyle\int_{\Omega^e} \boldsymbol{S}^T \rho\,\dot{u}\,\mathrm{d}x & 0 = \displaystyle\int_{\Omega^e} \rho \boldsymbol{S}^T \boldsymbol{S}\dot{\boldsymbol{U}}^e\,\mathrm{d}x & 0 = \boldsymbol{C}^e\dot{\boldsymbol{U}}^e \\
\quad + \displaystyle\int_{\Omega^e} \boldsymbol{S}^T \beta\,u\,\mathrm{d}x & \quad + \displaystyle\int_{\Omega^e} \beta \boldsymbol{S}^T \boldsymbol{S}\boldsymbol{U}^e\,\mathrm{d}x & \quad + \boldsymbol{K}^e_{\boldsymbol{\beta}}\boldsymbol{U}^e \\
\quad - \displaystyle\int_{\Omega^e} \boldsymbol{S}^T s\,\mathrm{d}x & \quad - \displaystyle\int_{\Omega^e} s \boldsymbol{S}^T\,\mathrm{d}x & \quad - \boldsymbol{F}^e \\
\quad + \displaystyle\int_{\Omega^e} \boldsymbol{S}^T \frac{\partial \bar{v}_x}{\partial x}\,\mathrm{d}x & \quad - \begin{Bmatrix} q_0 \\ q_1 \end{Bmatrix} & \\
 & \quad + \displaystyle\int_{\Omega^e} k_x \boldsymbol{G}^T \boldsymbol{G}\boldsymbol{U}^e\,\mathrm{d}x & \quad \boldsymbol{K}^e_{\boldsymbol{k}}\boldsymbol{U}^e
\end{array}
$$

Note that the nodal values $\boldsymbol{U}^e$ and $\dot{\boldsymbol{U}}^e$ can be moved outside the integral because they are constant in the space of the element.

Matrix formulation of the FEM (cont.)

The **element** matrices are:

$$\boldsymbol{C}^e = \int_{\Omega^e} \rho \boldsymbol{S}^T \boldsymbol{S} \, \mathrm{d}x$$

$$\boldsymbol{K}^{\boldsymbol{e}}_{\boldsymbol{\beta}} = \int_{\Omega^e} \beta \boldsymbol{S}^T \boldsymbol{S} \, \mathrm{d}x$$

$$\boldsymbol{K}^{\boldsymbol{e}}_{\boldsymbol{k}} = \int_{\Omega^e} k_x \boldsymbol{G}^T \boldsymbol{G} \boldsymbol{U}^e \, \mathrm{d}x$$

$$\boldsymbol{F}^e = \int_{\Omega^e} s \boldsymbol{S}^T \, \mathrm{d}x + \begin{Bmatrix} q_0 \\ q_1 \end{Bmatrix}$$

Therefore, the element system is:

$$\boxed{\boldsymbol{C}^e \dot{\boldsymbol{U}}^e + \boldsymbol{K}^e \boldsymbol{U}^e = \boldsymbol{F}^e}$$

with:

$$\boldsymbol{K}^e = \boldsymbol{K}^{\boldsymbol{e}}_{\boldsymbol{k}} + \boldsymbol{K}^{\boldsymbol{e}}_{\boldsymbol{\beta}}$$

Matrix formulation of the FEM (cont.)

Notes:

- ☐ Is not too difficult to extend the previous equations to multi-dimensions;
- ☐ Also, those equations are valid for any type of element;
- ☐ For **simple shape functions**, all integrals can be analytically evaluated. Otherwise, numerical integration (quadrature) can be employed;
- ☐ Some terms can be dropped for particular problems. For example, if there is no source, i.e. $s(x) = 0$, the corresponding term can be dropped. Also, if there is no rate-dependency, the term with matrix $\boldsymbol{C}^e$ can be dropped. Finally, if $\beta = 0$, matrix $\boldsymbol{K}^{\boldsymbol{e}}_{\boldsymbol{\beta}}$ does not need to be computed.

Matrix formulation of the FEM (cont.)

Let's then integrate those expressions for a linear element with 2 nodes, assuming that ρ, β, and k_x are constants. The first integral type is:

$$\begin{aligned}
\int_{\Omega^e} \boldsymbol{S}^T \boldsymbol{S} \, \mathrm{d}x &= \int_{x_0}^{x_1} \begin{bmatrix} \frac{x_1 - x}{\ell} \\ \frac{x - x_0}{\ell} \end{bmatrix} \begin{bmatrix} \frac{x_1 - x}{\ell} & \frac{x - x_0}{\ell} \end{bmatrix} \mathrm{d}x \\
&= \frac{1}{\ell^2} \begin{bmatrix} \int_{x_0}^{x_1} (x_1 - x)^2 \, \mathrm{d}x & \int_{x_0}^{x_1} (x_1 - x)(x - x_0) \, \mathrm{d}x \\ \int_{x_0}^{x_1} (x - x_0)(x_1 - x) \, \mathrm{d}x & \int_{x_0}^{x_1} (x - x_0)^2 \, \mathrm{d}x \end{bmatrix} \\
&= \frac{1}{\ell^2} \begin{bmatrix} \frac{\ell^3}{3} & \frac{\ell^3}{6} \\ \frac{\ell^3}{6} & \frac{\ell^3}{3} \end{bmatrix} = \frac{\ell}{6} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}
\end{aligned}$$

And the second one is:

$$\begin{aligned}
\int_{\Omega^e} \boldsymbol{G}^T \boldsymbol{G} \, \mathrm{d}x &= \int_{x_0}^{x_1} \begin{bmatrix} \frac{-1}{\ell} \\ \frac{1}{\ell} \end{bmatrix} \begin{bmatrix} \frac{-1}{\ell} & \frac{1}{\ell} \end{bmatrix} \mathrm{d}x \\
&= \frac{1}{\ell^2} \begin{bmatrix} \int_{x_0}^{x_1} 1 \, \mathrm{d}x & \int_{x_0}^{x_1} -1 \, \mathrm{d}x \\ \int_{x_0}^{x_1} -1 \, \mathrm{d}x & \int_{x_0}^{x_1} 1 \, \mathrm{d}x \end{bmatrix} \\
&= \frac{1}{\ell} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}
\end{aligned}$$

Matrix formulation of the FEM (cont.)

We can then write the element matrices as follows (for a 2-node linear element):

$$\boldsymbol{C}^e = \frac{\rho\,\ell}{6} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \qquad \boldsymbol{K}^e_{\boldsymbol{\beta}} = \frac{\beta\,\ell}{6} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \qquad \boldsymbol{K}^e_{\boldsymbol{k}} = \frac{k_x}{\ell} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

The integral including the source term $s(x)$ can only be integrated after a specific problem is defined. For example, suppose $s(x) = x$, then:

$$\begin{aligned}
\int_{\Omega^e} s \boldsymbol{S}^T \, \mathrm{d}x &= \int_{x_0}^{x_1} x \begin{bmatrix} \frac{x_1 - x}{\ell} \\ \frac{x - x_0}{\ell} \end{bmatrix} \mathrm{d}x \\
&= \frac{1}{\ell} \begin{bmatrix} \int_{x_0}^{x_1} (x_1\, x - x^2) \, \mathrm{d}x \\ \int_{x_0}^{x_1} (x^2 - x_0\, x) \, \mathrm{d}x \end{bmatrix} = \frac{\ell}{6} \begin{Bmatrix} 2\,x_0 + x_1 \\ x_0 + 2\,x_1 \end{Bmatrix}
\end{aligned}$$

or, if the source term is constant: $s(x) = c$, then:

$$\begin{aligned}
\int_{\Omega^e} s \boldsymbol{S}^T \, \mathrm{d}x &= c \int_{x_0}^{x_1} \begin{bmatrix} \frac{x_1 - x}{\ell} \\ \frac{x - x_0}{\ell} \end{bmatrix} \mathrm{d}x \\
&= \frac{c}{\ell} \begin{bmatrix} \int_{x_0}^{x_1} (x_1 - x) \, \mathrm{d}x \\ \int_{x_0}^{x_1} (x - x_0) \, \mathrm{d}x \end{bmatrix} = \frac{c\,\ell}{2} \begin{Bmatrix} 1 \\ 1 \end{Bmatrix}
\end{aligned}$$

Matrix formulation of the FEM: Summary

To summarize, in order to obtain the element equations using the **Galerkin finite element method**, we:

- □ Replace the weak form by several smaller integrals, each one corresponding to one element
- □ Assume a solution for the primary variable (such as $u(x)$) based on nodal values $\boldsymbol{U}^e$
- □ Apply the integration by parts to move the higher order derivatives to the shape functions and also to account for natural boundary conditions
- □ Integrate the weighted-residuals expressions for **one element** within its domain Ω^e; hence obtaining $\boldsymbol{C}^e$, $\boldsymbol{K}^e$, and $\boldsymbol{F}^e$

Later on, some **pre-derived** finite element equations for a number of PDEs will be directly presented – the assembly and solution process will not be changed though.

Computer implementation of the FEM solution

Python class: FEMmesh

Fundamentals
Weighted-Residuals
FEM: Introduction
FEM: Formulation
FEM: Implementation
▷ FEMmesh 1
FEMmesh 2
FEMmesh 3
Ediffusion1D 1
Ediffusion1D 2
Ediffusion1D 3
Ediffusion1D 4
Ediffusion1D 5
FEMsolver 1
FEM: Examples 1D

Before diving into the implementation of (finite) elements and a corresponding FE solver, a mesh structure is defined: **`FEMmesh`** class. This class contains all properties and methods required to represent and handle a finite element mesh.

To define a FE mesh, only two lists are required: (1) **`V`** holding all information on **Vertices**; and (2) **`C`** holding all information on **Cells**. These contain purely geometric information. Later on, **Nodes** and **Elements** can be constructed based on vertices and cells.

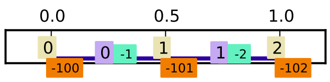
The conceptual difference between vertices/cells *versus* nodes/elements taken here is that the latter structures contain also material and simulation properties, such as solution variables (temperature, displacements, etc.) and element matrices (mass, stiffness, etc.).

Another important feature assigned to **`V`** and **`C`** is that they will contain **tags** indicating any geometric entity that later will be employed on the definition of **boundary conditions**. With 2D geometries, **vertices** and **edges** (sometimes called **faces**) can be tagged.

Python class: FEMmesh: examples

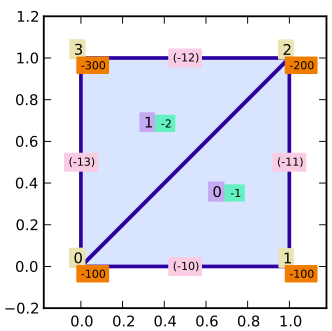
Fundamentals
Weighted-Residuals
FEM: Introduction
FEM: Formulation
FEM: Implementation
FEMmesh 1
▷ FEMmesh 2
FEMmesh 3
Ediffusion1D 1
Ediffusion1D 2
Ediffusion1D 3
Ediffusion1D 4
Ediffusion1D 5
FEMsolver 1
FEM: Examples 1D

1D mesh




```
from FEMmesh import * # FEM mesh class
#   vert_num  tag    x
V = [[ 0,    -100,  0.0], # vert # 0
     [ 1,    -101,  0.5], # vert # 1
     [ 2,    -102,  1.0]] # vert # 2
#   cell_num  tag  vertices
C = [[ 0,     -1,   [0,1]], # cell # 0
     [ 1,     -2,   [1,2]]] # cell # 1
m = FEMmesh(V, C) # construct a mesh object
m.draw()          # draw mesh using msys_drawmesh
m.show()          # or axis('equal'); show()
```

2D mesh



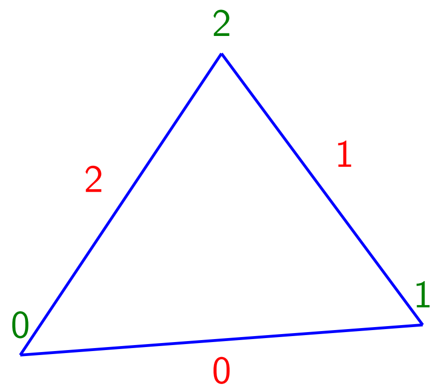

```
from FEMmesh import * # FEM mesh class
#   vert_num  tag    x     y
V = [[ 0,    -100,  0.0,  0.0], # vert # 0
     [ 1,    -100,  1.0,  0.0], # vert # 1
     [ 2,    -200,  1.0,  1.0], # vert # 2
     [ 3,    -300,  0.0,  1.0]] # vert # 3
#   cell_num  tag   vertices  edge_tags
C = [[ 0,     -1,   [0,1,2],  {0:-10, 1:-11}], # cell # 0
     [ 1,     -2,   [0,2,3],  {1:-12, 2:-13}]] # cell # 1
m = FEMmesh(V, C) # construct a mesh object
m.draw()          # draw mesh using msys_drawmesh
m.show()          # or axis('equal'); show()
```

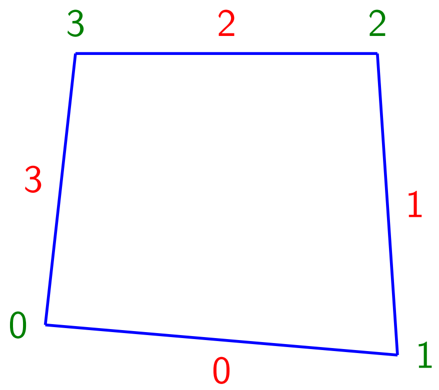
Python class: FEMmesh: edge/face tags

For 2D meshes, when defining cells in a sub-list of `C`, an optional dictionary can be specified, indicating what tags are attached to edges. For example, to attach tags to all edges of a quadrilateral:

```
C = [ ...
      [ncell, -1, [0,1,2,3], {0:-10, 1:-11, 2:-12, 3:-13}],
      ... ]
```

The numbering convention of vertices and edges is:





Python class: Ediffusion1D

The computer implementation of the FEM model for the 1D diffusion equation can be facilitated with the introduction of a Python class: **`Ediffusion1D`**. Additionally, this will be of great convenience when writing a FEM code that can handle other problems as well: 2D diffusion, elasticity, etc.

The first step is to create a file named `Ediffusion1D.py` containing two functions and one class as follows:

```
def info():                                 # returns all information corresponding to
    ...                                     #     a 1D diffusion problem
def alloc(verts, params):                   # allocates and returns an instance of
    ...                                     #     Ediffusion1D
class Ediffusion1D:                         # element class for 1D diffusion problems
    def __init__(self, verts, params):      # constructor initialised with a list of
        ...                                 #     vertices and a dictionary of parameters
    def set_nat_bcs(self, ...): pass        # not needed in this element
    def calc_C(self): ...                   # returns the element Ce matrix
    def calc_K(self): ...                   # returns the element Ke matrix
    def calc_F(self): ...                   # returns the element Fe vector
    def calc_secondary(self, Ue):           # calculates any secondary variables after
        ...                                 #     the element Ue vector has been found
```

Python class: Ediffusion1D – `info` and `alloc`

The function `info` will help the `FEMsolver` to set up all data required for a simulation and is a convenient way to gather information on a specific element **by name** before allocating any instance of this element. In other languages, this function is similar to a *static* method. Note: Python can handle static methods as well using *decorators*, but we will keep with this simple solution.

For our `Ediffusion1D` problem:

```python
def info():                  # information function
    ndim = 1                 # number of space dimensions => 1D
    nsov = 1                 # number of solution variables. only one => 'u' variable
    vbcs = {'u':(0,'utype'), # vertex boundary conditions info. 0 is the index of u
            'q':(0,'ftype')} #    also, 0 is the index of f, which is of 'ftype'
    ebcs   = {}              # edge/face boundary conditions info (no edges/faces here)
    secvs = ['wx']           # secondary variables. wx = - kx * dudx (seepage velocity)
    secgs = {'w':'wx'}       # secondary variables groups: 'w' has only one component: 'wx'
    return ndim, nsov, vbcs, ebcs, secvs, secgs # returns 6 variables holding all data
                                                #    required to set any simulation up

def alloc(verts, params):               # allocate one instance of Ediffusion1D
    return Ediffusion1D(verts, params)  # returns this instance
```


Python class: Ediffusion1D – `constructor`

The constructor of `Ediffusion1D` expects two variables: `verts` and `params`. The first one contains two rows of **V**, defining the two nodes of the element. The second one is a dictionary with all required and optional parameters. For example:

```python
def __init__(self, verts, params):
    """
    1D diffusion problem
    ====================

        Solving:         2
                du      d u
            rho == - kx ==== + beta*u = s(x)
                dt      d x2
        Example of input:
                 global_id  tag    x
            verts = [[3,    -100,  0.75],
                     [4,    -100,  1.0 ]]
            params = {'rho':1.0, 'beta':1.0, 'kx':1.0,
                      'source':src_val_or_fcn}
        Note:
            src_val_or_fcn: can be a constant value (float) or a callback
                            function such as lambda x: x**2.0
    """
```

Python class: Ediffusion1D – constructor (cont.)


```
def __init__(self, verts, params):                    # constructor of Ediffusion1D
    self.x0, self.x1    = verts[0][2], verts[1][2]  # left and right node coordinates
    self.l              = self.x1 - self.x0         # length of element (has to be positive)
    self.rho, self.beta = 0.0, 0.0                  # rho and beta default values
    self.kx             = params['kx']              # kx parameter. must be provided in params
    self.has_source      = False                    # has source term?
    if params.has_key('rho'):    self.rho  = params['rho']  # store 'rho',  if specified
    if params.has_key('beta'):   self.beta = params['beta'] # store 'beta', if specified
    if params.has_key('source'): self.source, self.has_source = params['source'], True
    cfc = self.rho  * self.l / 6.0                  # coefficient of C matrix
    cfb = self.beta * self.l / 6.0                  # coefficient of Kb matrix
    cfk = self.kx / self.l                          # coefficient of Kk matrix
    self.C = cfc * array([[2., 1.], [ 1., 2.]])     # Ce matrix
    self.K = cfb * array([[2., 1.], [ 1., 2.]]) \   # Ke matrix = Kk matrix
           + cfk * array([[1.,-1.], [-1., 1.]])     #          + Kbeta matrix
    self.Fs = zeros(2)                              # source term vector
    if self.has_source:                             # if it has source prescribed, then:
        if isinstance(self.source, float):          # has constant value source?
            cf = self.source * self.l / 2.0         # coefficient of Fs
            self.Fs = cf * array([1.0, 1.0])        # vector corresponding to source term
        else:           # source term is a function(x) => execute numerical integration
            i0 = lambda x: ((self.x1-x)/self.l) * self.source(x) # first integrand of Fs
            i1 = lambda x: ((x-self.x0)/self.l) * self.source(x) # second integrand of Fs
            self.Fs = array([quad(i0, self.x0, self.x1)[0],      # conduct numerical
                             quad(i1, self.x0, self.x1)[0]])     # integration using 'quad'
```


Python class: Ediffusion1D – methods

The other methods of `Ediffusion1D` are:

```
def set_nat_bcs(self, ...): pass               # set natural boundary conditions:
                                               #    not needed in this element because
                                               #    it does not have 'edges'/'faces'

def calc_C(self): return self.C                # returns the Ce matrix
def calc_K(self): return self.K                # returns the Ke matrix
def calc_F(self): return self.Fs               # returns the Fe vector

def calc_secondary(self, Ue):                  # calculates secondary data
    wx  = - self.kx * (Ue[1] - Ue[0]) / self.l # auxiliary variable (seepage velocity)
    ips = [((self.x0 + self.x1)/2.0 , )]       # (x,y) coordinates of the centre of element
                                               #    notice the comma after the x-coord.
                                               #    this is necessary to force a tuple

    svs = {'wx': [wx]}                         # store wx into a dictionary
    return ips, svs                            # returns the coordinates of all
                                               #    integration points (just one =>
                                               #    the centroid of the element)
                                               #    and all secondary variables in a
                                               #    dictionary: one 'wx' for each
                                               #    integration point (hence a list)
```

Python class: FEMsolver

Now, a `FEMsolver` class can be defined. It contains all routines to run a simulation and generate output files. The following methods are then defined:

```
class FEMsolver:
    def __init__(self, mesh, ename, params): # constructor
        """
        mesh:   an instance of FEMmesh
        ename:  element name. ex: 'Ediffusion1D'
        params: a dictionary connecting elements tags to a dictionary of
                element parameters. Ex:
                    params = { -1:{'rho':1.0, 'beta':0.0, 'kx':1.0},
                               -2:{'rho':2.0, 'beta':0.5, 'kx':1.5} }
        """
    def set_bcs(self, eb={}, vb={}):                           # set boundary conditions
    def solve_steady(self):                                    # solve steady/equilib problem
    def solve_eigen(self):                                     # run eigenvalue analysis
    def solve_transient(self, t0, U0, tf, dt, theta=0.5):     # solve transient problem
    def calc_secondary(self, method=2, calc_evalues=True):    # calc secondary variables
    def write_vtu(self, fnkey, onlymsh=False):                 # write .vtu file for ParaView
    def print_u(self, spaces=12, digits=6, numformat=None):   # print resulting u values
    def print_e(self, spaces=12, digits=6, numformat=None,    # print resulting secondary
                ncols=6, extrap=False):                        #     values at the centre of
                                                               #     elements
```


Examples of 1D solutions obtained with the finite element method

Example 1: First question of assignment 3 – convection along a rod

Suppose we want to solve:

$$-k_x \frac{\partial^2 u}{\partial x^2} + c_c (u - u_c) = 0$$

where $c_c = 2\pi$ and $u_c = 20$ are constants allowing the simulation of heat transfer via **c**onvection. c_c controls the convection transfer rate and u_c is the temperature of the surrounding ambient. The length of the bar is $L_x = 0.05$ and is divided into 2 elements.

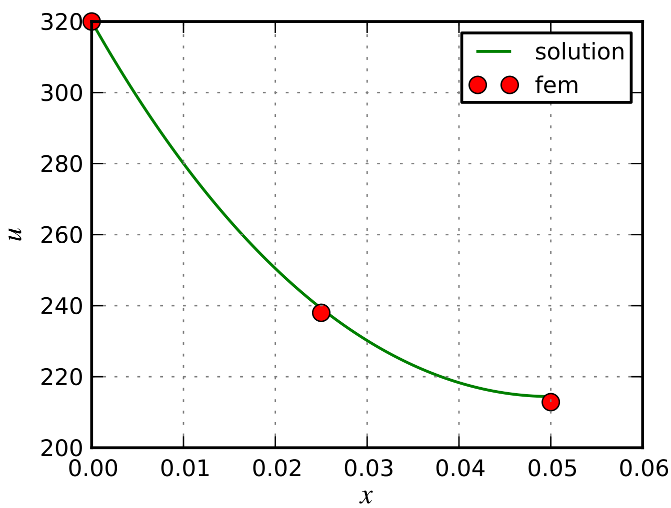
$k_x = 0.01571$ and the temperature to the left end is $u_L = 320$.

```
from FEMsolver import *     # the FEM solver
from msys_fig import *      # my routines for figures
V = [[0, -101, 0.0  ],      # list of vertices. 1st
     [1,    0, 0.025],      # second vertex
     [2,    0, 0.05 ]]      # third vertex
C = [[0, -1, [0,1]], [1, -1, [1,2]]] # cells
m = FEMmesh(V, C)                    # mesh object
kx, cc, uc = 0.01571, 2.*pi, 20.0    # parameters
p  = {-1:{'kx':kx, 'beta':cc, 'source':cc*uc}} # params
s  = FEMsolver(m, 'Ediffusion1D', p)  # solver
vb = {-101:{'u':320.}}      # boundary conditions
s.set_bcs(vb=vb)            # set boundary conditions
s.solve_steady()            # solve steady problem
s.print_u()                 # print results
def usol(x):                # closed-form solution
    return uc + (320.-uc) * cosh(sqrt(cc/kx)*(0.05-x))\
              / cosh(sqrt(cc/kx)*0.05)
X = [v[2] for v in m.V]     # all points
x = linspace(0.,0.05,101)   # many points
plot(x,usol(x),'g-',label='solution',clip_on=0)
plot(X,s.U,'ro',label='fem',clip_on=0)
Gll('x','u')                # grid, labels, legend
show()
```


Example 1: results

The output is:

```
================
node           u
----------------
   0  320.000000
   1  237.990766
   2  212.840996
================
```



Example 2: Temperature distribution along a rod with convection and source term

Solving:

$$-\frac{\partial^2 u}{\partial x^2} + u = 15\frac{\sinh(4\,x)}{\sinh 4} + x^2 - 2$$

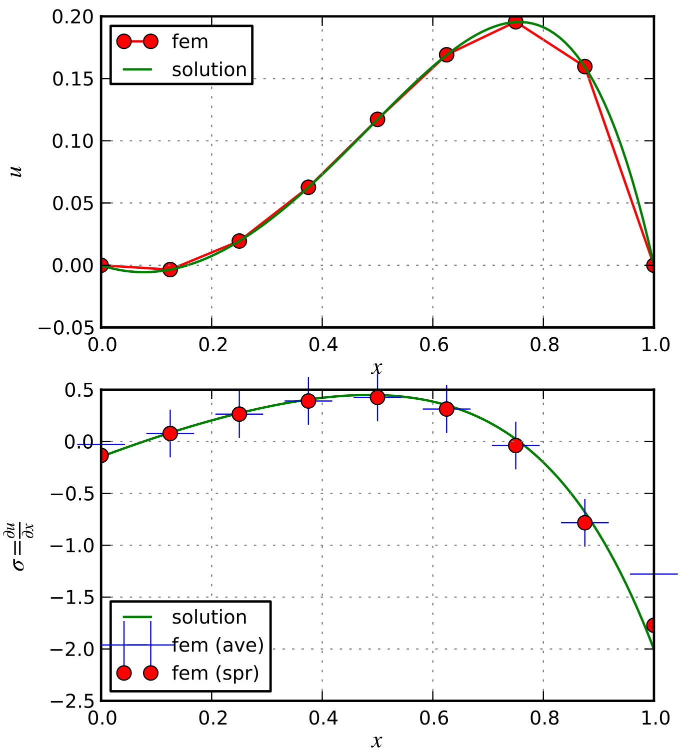
```
from FEMsolver import *
from numpy     import sinh
# generate mesh
Lx, nx = 1.0, 9           # length of bar, number of points
m      = Gen1Dmesh(Lx, nx) # convenience function for 1D meshes => will tag the first
                           #    vertex with -101 and the last one with -102
# parameters
p = {-1:{'beta':1., 'kx':1.,
         'source':lambda x: 15.*sinh(4.*x)/sinh(4.)+x**2.-2.}}
# allocate fem solver object
s = FEMsolver(m, 'Ediffusion1D', p)
# set boundary conditions
vb = {-101:{'u':0.0}, -102:{'u':0.0}} # set first and last vertices with 0
                                      #   temperature units
s.set_bcs(vb=vb) # vb = vertex boundary conditions
# solve steady problem
s.solve_steady()
# print results at nodes and elements
s.print_u()
s.print_e()
```


Example 2: results

The closed-form solution is:

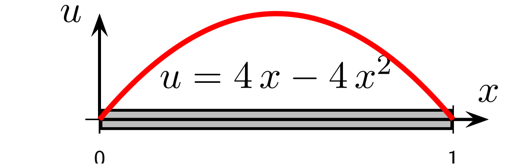
$$u_{sol} = x^2 - \frac{\sinh(4\,x)}{\sinh 4}$$

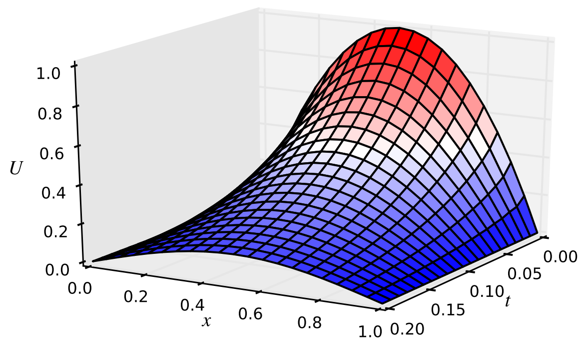
$$\frac{\mathrm{d}u_{sol}}{\mathrm{d}x} = 2\,x - \frac{4\,\cosh(4\,x)}{\sinh 4}$$



Example 3: Transient cooling of a hot bar with initial parabolic temperature distribution

Solution of an insulated rod of unit length with the two extremes exposed to 0 temperature units. $k_x = 1$. The initial temperature is distributed as shown below:

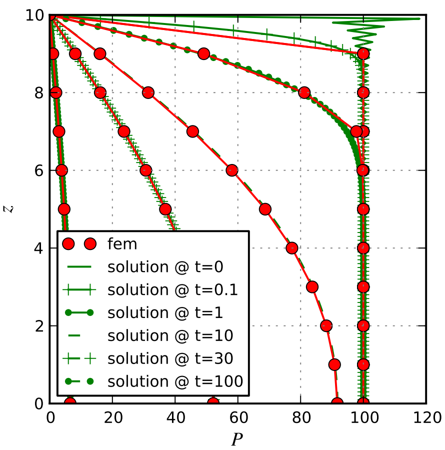





```
from FEMsolver import *
from msys_fig  import *
from femaux    import *
# generate 1D mesh
m  = Gen1Dmesh(1.0, 21) # Lx=1, nx=21
# create dictionary with all parameters
p = {-1:{'rho':1.0, 'kx':1.0}}
# allocate fem solver object
s = FEMsolver(m, 'Ediffusion1D', p)
# set boundary conditions
vb = {-101:{'u':0.}, -102:{'u':0.}}
s.set_bcs(vb=vb)
# initial time, final time, and time step
t0, tf, dt = 0.0, 0.2, 0.01
# all points coordinates
X = array([v[2] for v in m.V])
# initial conditions
U0 = 4.0*X-4.0*X**2.0
# solve transient problem
s.solve_transient(0.0, U0, tf, dt)
# plot
tt,xx = meshgrid(s.Tout, X)
uu = Uout2Umat(s)
ax = PlotSurf(tt,xx,uu,'t','x','u',0.,1.)
ax.view_init(20.,30.); show()
```


Example 4: consolidation of a half-drained soil layer

Question 2 of Assignment 3: the bottom of the column is impermeable and a pressure $P = 0$ is applied at the top. The height is $H = 10$ and the coefficient of consolidation is $c_v = 1.2$.




```
from FEMsolver import *
from msys_fig  import *
# constants
cv = 1.2   # coefficient of consolidation
P0 = 100.0 # increment of pore water pressure
# mesh
m  = Gen1Dmesh(10.0, 11) # Lx, nx
# parameters and solver
p  = {-1:{'rho':1., 'kx':cv}}
s  = FEMsolver(m, 'Ediffusion1D', p)
# boundary conditions
s.set_bcs(vb={-102:{'u':0.}})
# initial time, final time, and time step
t0, tf, dt = 0., 100., 0.4
# initial conditions. solve transient problem
U0 = P0*ones(m.nv);  U0[-1] = 0.
s.solve_transient(t0, U0, tf, dt)
# plot
X = [v[2] for v in m.V] # all points
for i, tout in enumerate([0., 1.0, 10.0, 30.0, 100.0]):
    iout = int(tout / dt)
    U = [s.Uout['u'][n][iout] for n in range(m.nv)]
    plot(U, X, label='time = %g'%tout, clip_on=False)
Gll('P','z', leg_loc='lower left');  show()
```

Numerical Methods in Engineering – Part IV

Dr Dorival Pedroso

June 19, 2012

A Word on Units

A word on units

Any units can be used in a FEM simulation as long as they are consistent. For instance, according to the **SI** system:

$$\underbrace{F}_{force} = \underbrace{m}_{mass} \times \underbrace{a}_{accel} \quad \Rightarrow \quad [\mathrm{kg}][\mathrm{m}][\mathrm{s}^{-2}] = [\mathrm{N}]$$

$$\underbrace{P}_{pressure/stress} = \underbrace{F}_{force} / \underbrace{A}_{area} \quad \Rightarrow \quad [\mathrm{N}][\mathrm{m}^{-2}] = [\mathrm{Pa}]$$

$$\underbrace{\rho}_{density} = \underbrace{m}_{mass} / \underbrace{V}_{volume} \quad \Rightarrow \quad [\mathrm{kg}][\mathrm{m}^{-3}]$$

If it is desired to work with $[\,\mathrm{MPa}]$ and $[\,\mathrm{m}]$, for example, then the other units must be changed as well. In this case, the change is:

$$F = P\,A \quad \Rightarrow \quad [\mathrm{MPa}][\mathrm{m}^2] = [\mathrm{MN}]$$

$$m = F/a \quad \Rightarrow \quad [\mathrm{MN}][\mathrm{m}^{-1}][\mathrm{s}^2] = 10^6[\mathrm{N}][\mathrm{m}^{-1}][\mathrm{s}^2] = 10^6[\mathrm{kg}] = [\mathrm{Gg}]$$

$$\rho = m/V \quad \Rightarrow \quad [\mathrm{Gg}][\mathrm{m}^{-3}]$$

A word on units

To use $[\,\mathrm{kPa}]$ and $[\,\mathrm{m}]$, the other units are:

$$F = P\,A \quad \Rightarrow \quad [\mathrm{kPa}][\mathrm{m}^2] = [\mathrm{kN}]$$

$$m = F/a \quad \Rightarrow \quad [\mathrm{kN}][\mathrm{m}^{-1}][\mathrm{s}^2] = 10^3[\mathrm{N}][\mathrm{m}^{-1}][\mathrm{s}^2] = [\mathrm{Mg}]$$

$$\rho = m/V \quad \Rightarrow \quad [\mathrm{Mg}][\mathrm{m}^{-3}]$$

Another more common choice is the use of $[\,\mathrm{N}]$ for force and $[\,\mathrm{mm}]$ for length. In this case:

$$a \quad \Rightarrow \quad [\mathrm{mm}][\mathrm{s}^{-2}]$$

$$P = F/A \quad \Rightarrow \quad [\mathrm{N}][\mathrm{mm}^{-2}] = 10^6[\mathrm{N}][\mathrm{m}^{-2}] = [\mathrm{MPa}]$$

$$m = F/a \quad \Rightarrow \quad [\mathrm{N}][\mathrm{mm}^{-1}][\mathrm{s}^2] = 10^3[\mathrm{N}][\mathrm{m}^{-1}][\mathrm{s}^2] = [\mathrm{Mg}]$$

$$\rho = m/V \quad \Rightarrow \quad [\mathrm{Mg}][\mathrm{mm}^{-3}]$$

Table 1: Some consistent units for FE analyses.

	standard	set # 1	set # 2	set # 3
time	s	s	s	s
length	m	m	m	mm
force	N	kN	MN	N
pressure/stress	Pa	kPa	MPa	MPa
mass	kg	Mg	Gg	Mg
density	kg/m^3	Mg/m^3	Gg/m^3	Mg/mm^3

Table 2: Example of material properties (approximated).

	E [MPa]	ν [−]	ρ [Gg/m^3]
Wood: Douglas-fir	13100	0.29	4.70×10^{-4}
Concrete: Low strength	22100	0.15	2.38×10^{-3}
Aluminum: 2014-T6	73100	0.35	2.79×10^{-3}
Steel: structural A36	200000	0.32	7.85×10^{-3}

Finite Element Method: Assembly of Element Equations and Solution of the Global Linear System

Assembly of element equations

The space discretisation of partial differential equations (PDEs) using the finite element method (FEM) results on a number of matrices and vectors. Each element generates a local system as follows:

$$\boldsymbol{K}^e\boldsymbol{U}^e = \boldsymbol{F}^e \qquad \textbf{Steady/Equilibrium problems}$$

$$\boldsymbol{C}^e\dot{\boldsymbol{U}}^e + \boldsymbol{K}^e\boldsymbol{U}^e = \boldsymbol{F}^e \qquad \textbf{Transient problems}$$

$$\boldsymbol{M}^e\ddot{\boldsymbol{U}}^e + \boldsymbol{C}^e\dot{\boldsymbol{U}}^e + \boldsymbol{K}^e\boldsymbol{U}^e = \boldsymbol{F}^e \qquad \textbf{Dynamics problems}$$

These have to be added, one by one, resulting into **global systems**:

$$\boldsymbol{K}\boldsymbol{U} = \boldsymbol{F} \qquad \textbf{Steady/Equilibrium problems}$$

$$\boldsymbol{C}\dot{\boldsymbol{U}} + \boldsymbol{K}\boldsymbol{U} = \boldsymbol{F} \qquad \textbf{Transient problems}$$

$$\boldsymbol{M}\ddot{\boldsymbol{U}} + \boldsymbol{C}\dot{\boldsymbol{U}} + \boldsymbol{K}\boldsymbol{U} = \boldsymbol{F} \qquad \textbf{Dynamics problems}$$

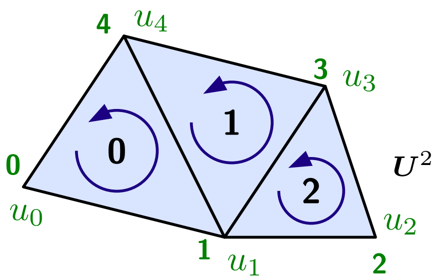
The process of adding element matrices into global matrices is known as **assembly** due to historical reasons such as the FEM being initially developed by structural engineers.

Assembly of element equations

Let's exemplify the assembly process for a $\boldsymbol{K}\boldsymbol{U} = \boldsymbol{F}$ system using triangles with one **solution variable** *per* node (`nsov=1`). These *solution variables* are also known as *degrees of freedom*; again due to historical heritage from structural mechanics.

The following mesh is considered:

$$\boldsymbol{U}^1 = \begin{Bmatrix} u_1 \\ u_3 \\ u_4 \end{Bmatrix} \qquad \boldsymbol{U}^0 = \begin{Bmatrix} u_0 \\ u_1 \\ u_4 \end{Bmatrix}$$



$$\boldsymbol{U}^2 = \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \end{Bmatrix}$$

Thus, the global system will have the following form:

$$\begin{bmatrix} \bigcirc & \bigcirc & \bigcirc & \bigcirc & \bigcirc \\ \bigcirc & \bigcirc & \bigcirc & \bigcirc & \bigcirc \\ \bigcirc & \bigcirc & \bigcirc & \bigcirc & \bigcirc \\ \bigcirc & \bigcirc & \bigcirc & \bigcirc & \bigcirc \\ \bigcirc & \bigcirc & \bigcirc & \bigcirc & \bigcirc \end{bmatrix} \begin{Bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \end{Bmatrix} = \begin{Bmatrix} \bigcirc \\ \bigcirc \\ \bigcirc \\ \bigcirc \\ \bigcirc \end{Bmatrix}$$

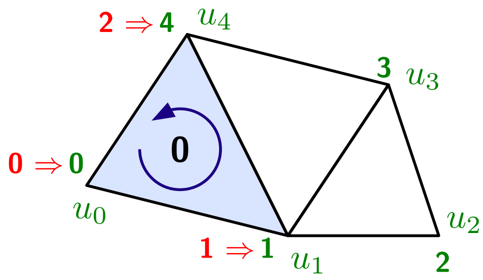
Assembly of element equations

For element 0, with **connectivity** $C_0 = \begin{bmatrix} 0 & 1 & 4 \end{bmatrix}$, the **local** stiffness matrix has the following form:

$$\begin{array}{cc} & \begin{array}{ccc} u_0 & u_1 & u_4 \\ \downarrow & \downarrow & \downarrow \end{array} \\ \boldsymbol{K}^0 = & \begin{bmatrix} 0_{00} & 0_{01} & 0_{04} \\ 0_{10} & 0_{11} & 0_{14} \\ 0_{40} & 0_{41} & 0_{44} \end{bmatrix} \begin{array}{l} \leftarrow u_0 \\ \leftarrow u_1 \\ \leftarrow u_4 \end{array} \end{array}$$

And the **local** $\boldsymbol{F}^e$ vector has the following form:

$$\boldsymbol{F}^0 = \begin{Bmatrix} 0_0 \\ 0_1 \\ 0_4 \end{Bmatrix} \begin{array}{l} \leftarrow u_0 \\ \leftarrow u_1 \\ \leftarrow u_4 \end{array}$$



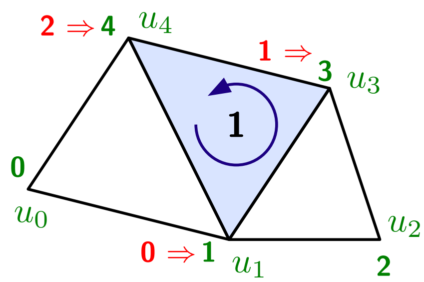
Assembly of element equations

For element 1, with **connectivity** $C_1 = \begin{bmatrix} 1 & 3 & 4 \end{bmatrix}$, the **local** stiffness matrix has the following form:

$$\begin{array}{cc} & \begin{array}{ccc} u_1 & u_3 & u_4 \\ \downarrow & \downarrow & \downarrow \end{array} \\ \boldsymbol{K}^1 = & \begin{bmatrix} 1_{11} & 1_{13} & 1_{14} \\ 1_{31} & 1_{33} & 1_{34} \\ 1_{41} & 1_{43} & 1_{44} \end{bmatrix} \begin{array}{l} \leftarrow u_1 \\ \leftarrow u_3 \\ \leftarrow u_4 \end{array} \end{array}$$

And the **local** $\boldsymbol{F}^e$ vector has the following form:

$$\boldsymbol{F}^1 = \begin{Bmatrix} 1_1 \\ 1_3 \\ 1_4 \end{Bmatrix} \begin{array}{l} \leftarrow u_1 \\ \leftarrow u_3 \\ \leftarrow u_4 \end{array}$$

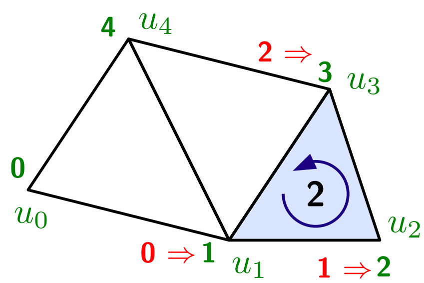


For element 2, with **connectivity** $C_2 = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$, the **local** stiffness matrix has the following form:

$$\boldsymbol{K}^2 = \begin{array}{c c} \begin{array}{ccc} u_1 & u_2 & u_3 \\ \downarrow & \downarrow & \downarrow \end{array} & \\ \begin{bmatrix} 2_{11} & 2_{12} & 2_{13} \\ 2_{21} & 2_{22} & 2_{23} \\ 2_{31} & 2_{32} & 2_{33} \end{bmatrix} & \begin{array}{l} \leftarrow u_1 \\ \leftarrow u_2 \\ \leftarrow u_3 \end{array} \end{array}$$

And the **local** $\boldsymbol{F}^e$ vector has the following form:

$$\boldsymbol{F}^2 = \begin{Bmatrix} 2_1 \\ 2_2 \\ 2_3 \end{Bmatrix} \begin{array}{l} \leftarrow u_1 \\ \leftarrow u_2 \\ \leftarrow u_3 \end{array}$$



Adding contributions from all elements to the global $\boldsymbol{K}$ matrix:

$$\boldsymbol{K} = \begin{array}{c c} \begin{array}{ccccc} u_0 & u_1 & u_2 & u_3 & u_4 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \end{array} & \\ \begin{bmatrix} 0_{00} & 0_{01} & & & 0_{04} \\ 0_{10} & 0_{11}+1_{11}+2_{11} & 2_{12} & 1_{13}+2_{13} & 0_{14}+1_{14} \\ & 2_{21} & 2_{22} & 2_{23} & \\ & 1_{31}+2_{31} & 2_{32} & 1_{33}+2_{33} & 1_{34} \\ 0_{40} & 0_{41}+1_{41} & & 1_{43} & 0_{44}+1_{44} \end{bmatrix} & \begin{array}{l} \leftarrow u_0 \\ \leftarrow u_1 \\ \leftarrow u_2 \\ \leftarrow u_3 \\ \leftarrow u_4 \end{array} \end{array}$$

Similarly, adding contributions from all elements to the global $\boldsymbol{F}$ vector:

$$\boldsymbol{F} = \begin{Bmatrix} 0_0 \\ 0_1+1_1+2_1 \\ 2_2 \\ 1_3+2_3 \\ 0_4+1_4 \end{Bmatrix} \begin{array}{l} \leftarrow u_0 \\ \leftarrow u_1 \\ \leftarrow u_2 \\ \leftarrow u_3 \\ \leftarrow u_4 \end{array}$$

Assembly of element equations

For some problems, the number of solution variables per node ($\mathtt{nsov}$) can be greater than one. For instance, in 2D problems of elasticity, each node has 2 variables: u_x and u_y, corresponding to the displacements at nodes – also known as degrees of freedom (dofs).

A convenient way to organise these variables in the *global* vector of unknowns $\boldsymbol{U}$ is illustrated to the right including all variables (or equations) to be solved for – all degrees of freedom. Each equation number (I) can be easily calculated as follows:

$$I = n \times \mathtt{nsov} + i$$

where i is the local $\mathtt{sov}$ index, n is the node number, and I the equation/global number. If the equation number I is given, the node and $\mathtt{sov}$ indices can also be recovered (integer division):

$$n = I \,/\, \mathtt{nsov} \qquad \text{and} \qquad i = I \,\%\, \mathtt{nsov}$$

$$\begin{array}{ccc} n & i & I \\ \left\{\begin{array}{c} u_{x0} \\ u_{y0} \\ u_{x1} \\ u_{y1} \\ u_{x2} \\ u_{y2} \\ u_{x3} \\ u_{y3} \\ u_{x4} \\ u_{y4} \\ u_{x5} \\ u_{y5} \end{array}\right\} & \begin{array}{c} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{array} & \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \end{array} \end{array}$$

$n \Rightarrow$ node
$i \Rightarrow \mathtt{sov}$
$I \Rightarrow$ global equation ($\mathtt{eq}$)

Assembly of element equations

The assembly procedure can be easily *automatised* after the introduction of an **assembly map** ($\mathtt{amap}$). For instance, this map can be created with the following Python script:

```python
# for a given mesh 'm':
nsov = 1                          # number of solution variables per node
amap = []                         # assembly map: list of location arrays for all elements
for c in m.C:                     # for each cell in mesh 'm'
    verts = [m.V[i] for i in c[2]] # collect vertices of cell 'c'
    loc   = []                    # location list
    for v in verts:               # for each vertex in cell 'c'
        n = v[0]                  # global node number
        for idx in range(nsov):   # for each index of solution variable
            eq = n * nsov + idx   # equation number
            loc.append(eq)        # save location list
    amap.append(loc)              # add location list to amap
```

For example, for the previous mesh:

```python
# with nsov == 1
amap = [[0, 1, 4], [1, 3, 4], [1, 2, 3]]

# with nsov == 2
amap = [[0, 1, 2, 3, 8, 9], [2, 3, 6, 7, 8, 9], [2, 3, 4, 5, 6, 7]]
```

Assembly of element equations

Units
FEM Assembly
Assembly 1
Assembly 2
Assembly 3
Assembly 4
Assembly 5
Assembly 6
Assembly 7
Assembly 8
▷ Assembly 9
Solution 1
Solution 2
Summary 1
Summary 2
Plane Trusses
Plane Frames
2D Diffusion Tri
Stress/Strain
2D Elastic Tri
Summary

Then, the assembly could be carried out with a code similar to:

```
for ie, e in enumerate(elems):            # for each element (e) with index (ie)
    Ke, Fe = e.calc_K(), e.calc_F()       # call element for Ke and Fe
    for i, I in enumerate(amap[ie]):      # for each local (i) and global (I) indices
        for j, J in enumerate(amap[ie]):  # for each local (j) and global (J) indices
            K[I,J] += Ke[i,j]             # add from Ke to K
        F[I] += Fe[i]                     # add from Fe to F
```

However, it's better to directly assemble only the parts $\boldsymbol{K}_{11}$ and $\boldsymbol{K}_{12}$ of the partitioned system, readily allowing the handling of essential boundary conditions.

```
for ie, e in enumerate(elems):            # for each element (e) with index (ie)
    Ke, Fe = e.calc_K(), e.calc_F()       # call element for Ke and Fe
    for i, I in enumerate(amap[ie]):      # for each local (i) and global (I) indices
        for j, J in enumerate(amap[ie]):  # for each local (i) and global (I) indices
            if   (I in eq1) and (J in eq1): K11[I,J] += Ke[i,j] # eq1 => eqs to be solved for
            elif (I in eq1) and (J in eq2): K12[I,J] += Ke[i,j] # eq2 => unknown equations
        if I in eq1: F[I] += Fe[i] # if (I) is in the list of equations to be solved for (eq1)
```

Note that the assembly process for the other matrices $\boldsymbol{M}$ and $\boldsymbol{C}$ is similar.

Solution of the global linear system

Units
FEM Assembly
Assembly 1
Assembly 2
Assembly 3
Assembly 4
Assembly 5
Assembly 6
Assembly 7
Assembly 8
Assembly 9
▷ Solution 1
Solution 2
Summary 1
Summary 2
Plane Trusses
Plane Frames
2D Diffusion Tri
Stress/Strain
2D Elastic Tri
Summary

The partitioned global system is:

$$\begin{bmatrix} \boldsymbol{K}_{11} & \boldsymbol{K}_{12} \\ \boldsymbol{K}_{21} & \boldsymbol{K}_{22} \end{bmatrix} \begin{Bmatrix} \boldsymbol{U}_1 ? \\ \boldsymbol{U}_2 \checkmark \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{F}_1 \checkmark \\ \boldsymbol{F}_2 ? \end{Bmatrix}$$

where the essential boundary conditions can easily be introduced. Here, $\boldsymbol{U}_2$ corresponds to the known or prescribed essential values and $\boldsymbol{U}_1$ indicates all equations that need to be solved for.

Note that essential values are always prescribed at nodes. Later on, a convenient method is introduced where tagged edges can carry on the prescribed essential values to the corresponding nodes.

Now, we have to solve:

$$\boldsymbol{U}_1 = \boldsymbol{K}_{11}{}^{-1}(\boldsymbol{F}_1 - \boldsymbol{K}_{12}\boldsymbol{U}_2)$$

If needed (not essentially required), the $\boldsymbol{F}_2$ vector, which implies the *reaction forces*, can be (post-)calculated with:

$$\boldsymbol{F}_2 = \boldsymbol{K}_{21}\boldsymbol{U}_1 + \boldsymbol{K}_{22}\boldsymbol{U}_2$$

Solution of the global linear system

When using a *sparse* solver – which is able to handle large sparse matrices – it is convenient to modify (or augment) the part $\boldsymbol{K}_{11}$ instead of removing the unnecessary rows and columns from this matrix (as we did in the FDM case). This modification is simply the placing of ones at the diagonal positions corresponding to prescribed essential values. Because the matrix $\boldsymbol{K}_{11}$ is assembled as a sparse structure, this placing will just make sure the matrix is invertible. Therefore:

$$\underbrace{\begin{bmatrix} \boldsymbol{K}_{11} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{1} \end{bmatrix}}_{\bar{\boldsymbol{K}}_{11} \Rightarrow \text{ augmented}} \underbrace{\begin{Bmatrix} \boldsymbol{U}_1 \\ \boldsymbol{U}_2 \end{Bmatrix}}_{\boldsymbol{U}} = \underbrace{\begin{Bmatrix} \boldsymbol{F}_1 - \boldsymbol{K}_{12}\boldsymbol{U}_2 \\ \boldsymbol{U}_2 \end{Bmatrix}}_{\boldsymbol{W} \Rightarrow \text{ workspace}}$$

After the solution the (whole) $\boldsymbol{U}$ vector is obtained:

$$\boldsymbol{U} = \left(\bar{\boldsymbol{K}}_{11}\right)^{-1} \boldsymbol{W}$$

Solution of the global linear system: Summary

Units
FEM Assembly
Assembly 1
Assembly 2
Assembly 3
Assembly 4
Assembly 5
Assembly 6
Assembly 7
Assembly 8
Assembly 9
Solution 1
Solution 2
▷ Summary 1
Summary 2
Plane Trusses
Plane Frames
2D Diffusion Tri
Stress/Strain
2D Elastic Tri
Summary

To summarise, the assembly and solution processes in the finite element method are based on:

- ☐ The assembly corresponds to the addition of finite terms that arise from the application of weighted-residuals in the element level
- ☐ A key concept during the assembly is the definition of **maps** connecting local nodal indices to global ones
- ☐ The use of sparse linear solvers largely speeds simulations up
- ☐ When using sparse solvers, the assembly process needs only to consider non-zero matrices that can directly be added to the matrices $\boldsymbol{K}_{11}$ and $\boldsymbol{K}_{12}$

Solution of the global linear system

- In structural problems, to each fixed degree of freedom corresponds a **reaction** force – e.x: R_{ux} or R_{uy}; reactions against u_x and u_y, respectively
- To find these reactions, we just have to subtract $\boldsymbol{F_2}$ from the prescribed nodal loads, including those that arise from the distributed loads
- For equilibrium, the sum of reactions must be equal and opposite in sign to the sum of applied loads $\Rightarrow$ quick and simple method to check consistency
- The concept of *equilibrium* is also applicable to diffusion problems, where the total heat or fluid flowing into the body must be equal to the one going out of the body

Plane Trusses $\Rightarrow$ `EelasticRod`

Plane Trusses ⇒ EelasticRod

Units
FEM Assembly
Plane Trusses
▷ EelasticRod 1
EelasticRod 2
EelasticRod 3
EelasticRod 4
EelasticRod 5
EelasticRod 6
EelasticRod 7
Example 1 a
Example 1 b
Example 1 c
Plane Frames
2D Diffusion Tri
Stress/Strain
2D Elastic Tri
Summary

Structural frameworks such as trusses can be easily solved with the finite element method. To do so, the method is applied to the equilibrium (differential) equations. Considering first the axial deformation of an elastic bar, the corresponding governing PDE is:

$$\rho\, A \frac{\partial^2 u_a}{\partial t^2} - E\, A \frac{\partial^2 u_a}{\partial x^2} = 0$$

where ρ is the material density, A is the cross-section area (constant along the bar), E is the Young modulus (constant along the bar) and u_a is the axial displacement.

The assumptions of small strains and that the stresses are uniform on any cross section are made here.

The FE method can then be derived after the development of the weak form of the previous equation.

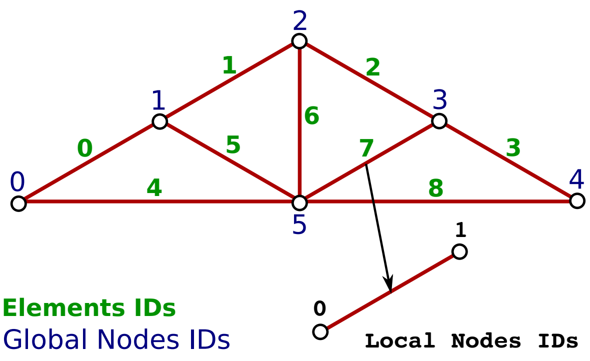
After obtaining the linear system of equations in a local (axial) form such as $\boldsymbol{K}_l^e \boldsymbol{U}_l^e = \boldsymbol{F}_l^e$, the element system can be calculated by means of a rotation transformation to the global system $x - y$.

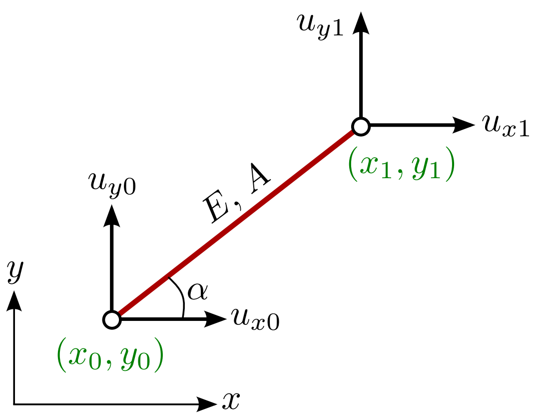
Plane Trusses ⇒ EelasticRod

Units
FEM Assembly
Plane Trusses
EelasticRod 1
▷ EelasticRod 2
EelasticRod 3
EelasticRod 4
EelasticRod 5
EelasticRod 6
EelasticRod 7
Example 1 a
Example 1 b
Example 1 c
Plane Frames
2D Diffusion Tri
Stress/Strain
2D Elastic Tri
Summary

The following additional characteristics are of relevance when applying the finite element method to this problem:

- Each element corresponds to a *rod* of linear elastic material; these resist axial forces only and are pin joined
- There are 2 solution variables/degrees of freedom per node: u_x and u_y; hence each element has 4 nodal variables





Plane Trusses ⇒ `EelasticRod`: element equations

The steady/equilibrium version of the **element equations** for a 2D rod represented by a **linear elastic material model** is:

$$\boldsymbol{K}^e \boldsymbol{U}^e = \boldsymbol{F}^e \qquad \text{with} \qquad \boldsymbol{U}^e = \begin{Bmatrix} u_{x0} \\ u_{y0} \\ u_{x1} \\ u_{y1} \end{Bmatrix}$$

and it can be shown that:

$$\boldsymbol{K}^e = \boldsymbol{T}^T \boldsymbol{K}_l^e \boldsymbol{T}$$

with:

$$\boldsymbol{K}_l^e = \frac{E\,A}{l}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \qquad \text{and} \qquad \boldsymbol{T} = \begin{bmatrix} c & s & 0 & 0 \\ 0 & 0 & c & s \end{bmatrix}$$

where E is the Young's (elastic) modulus, A is the cross-sectional area, l is the rod length, $c = \cos\alpha$, and $s = \sin\alpha$.

Plane Trusses ⇒ `EelasticRod`: element equations

The auxiliary variables c and s can be calculated as follows:

$$l = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2} \qquad c = \frac{x_1 - x_0}{l} \qquad s = \frac{y_1 - y_0}{l}$$

Note that the equations of the form $\boldsymbol{K}^e \boldsymbol{U}^e = \boldsymbol{F}^e$ are for static (or steady) problems and that $\boldsymbol{K}^e$ is known as **stiffness matrix**.

For dynamics problems, the element equations become:

$$\boldsymbol{M}^e \ddot{\boldsymbol{U}}^e + \boldsymbol{K}^e \boldsymbol{U}^e = \boldsymbol{F}^e$$

with:

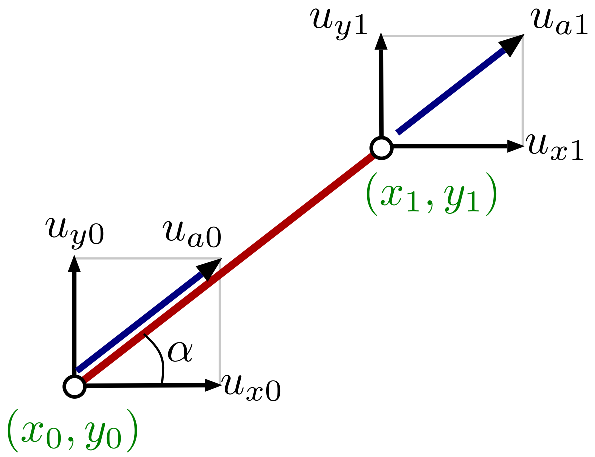
$$\boldsymbol{M}^e = \boldsymbol{T}^T \boldsymbol{M}_l^e \boldsymbol{T} \qquad \text{and} \qquad \boldsymbol{M}_l^e = \frac{\rho\,A\,l}{6}\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

where $\boldsymbol{M}^e$ is known as **mass matrix** and ρ is the material density.

Plane Trusses ⇒ `EelasticRod`: post-computations

For plane trusses, the following quantities are of great interest, especially when designing structures: axial strain ε_a, axial stress σ_a, and axial force N.

To calculate these values, **after** the primary values $\boldsymbol{U}^e$ have been found, the axial displacements u_{a0} and u_{a1} have to be computed first.



Plane Trusses ⇒ `EelasticRod`: post-computations

These axial displacements can be easily calculated as follows:

$$\underbrace{\begin{Bmatrix} u_{a0} \\ u_{a1} \end{Bmatrix}}_{\boldsymbol{U}_a} = \underbrace{\begin{bmatrix} c & s & 0 & 0 \\ 0 & 0 & c & s \end{bmatrix}}_{\boldsymbol{T}} \underbrace{\begin{Bmatrix} u_{x0} \\ u_{y0} \\ u_{x1} \\ u_{y1} \end{Bmatrix}}_{\boldsymbol{U}^e}$$

or $\boldsymbol{U}_a = \boldsymbol{T}\boldsymbol{U}^e$, where $\boldsymbol{T}$ is a "transformation" (rotation) matrix with $c = \cos\alpha$ and $s = \sin\alpha$.
Then we can calculate the following **secondary variables**:

$$\varepsilon_a = (u_{a1} - u_{a0})/l \qquad \text{axial strain}$$
$$\sigma_a = E\,\varepsilon_a \qquad \text{axial stress}$$
$$N = A\,\sigma_a \qquad \text{axial force}$$

where l is the length or the rod, E the Young modulus, and A the cross-sectional area.

Plane Trusses ⇒ EelasticRod: Python class

The information provided by EelasticRod is:

```python
def info():
    ndim = 2 # number of space dimensions
    nsov = 2 # number of solution variables (2 => ux, uy)
    # vertex boundary conditions info
    vbcs = {'ux':(0,'utype'), # x-displacement
            'uy':(1,'utype'), # y-displacement
            'fx':(0,'ftype'), # x-force
            'fy':(1,'ftype')} # y-force
    ebcs  = {}                # edge boundary conditions info
    secvs = ['ea', 'sa', 'N'] # axial strain, stress and normal force
    secgs = {'ea':'ea', 'sa':'sa', 'N':'N'} # secondary variables groups
    return ndim, nsov, vbcs, ebcs, secvs, secgs
```

and its constructor requires:

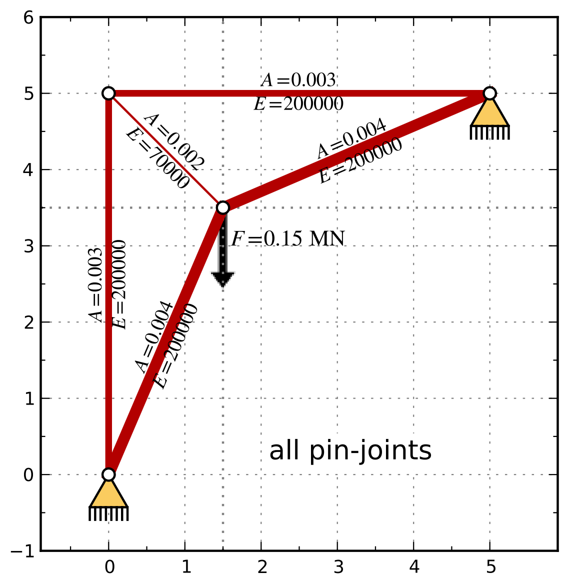
```python
class EelasticRod:
    def __init__(self, verts, params):
        """
                    global_id  tag   x     y
            verts = [[3,     -100, 0.75, 0.0],
                     [4,     -100, 1.0,  0.0]]
            params = {'E':1.0, 'A':1.0, 'rho':1.0}
        """
```


Plane Trusses ⇒ EelasticRod: example 1

Example from [Bhatti 2005; p25; Example 1.4]

For the plane truss shown to the right, find the nodal displacements and member stresses at equilibrium state. The five steps required are:

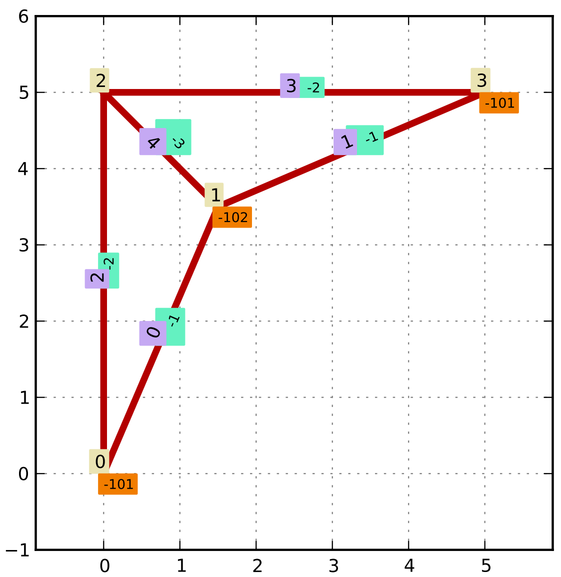
```python
from FEMsolver import *
# 1) input mesh
V = [[0, -101, 0.0, 0.0], # <= id tag x y
     [1, -102, 1.5, 3.5], # lengths
     [2,    0, 0.0, 5.0], #   in [m]
     [3, -101, 5.0, 5.0]]
C = [[0,  -1, [0,1]], # <= id tag verts
     [1,  -1, [1,3]],
     [2,  -2, [0,2]],
     [3,  -2, [2,3]],
     [4,  -3, [2,1]]]
m = FEMmesh(V, C)
m.draw(); m.show()
# 2) create dictionary with all parameters
p = {-1:{'E':200000.0, 'A':0.004}, # E: [MPa]
     -2:{'E':200000.0, 'A':0.003}, # A: [m2]
     -3:{'E': 70000.0, 'A':0.002}}
```



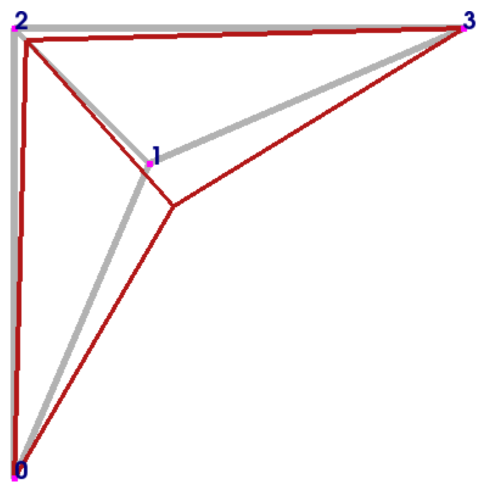
Units: set # 2 in Tab. 1 ⇒ length in m, A in m^2 and E in MPa

```
# 3) allocate fem solver object
s = FEMsolver(m, 'EelasticRod', p)

# 4) set boundary conditions
vbcs = {-101:{'ux':0.0,'uy':0.0},
        -102:{'fy':-0.15}}      # fy: [MN]
s.set_bcs(vb=vbcs)     # vertex bry conds

# 5) solve equilibrium problem
calc_reactions = True
s.solve_steady(calc_reactions)

# 6) print results
s.print_u() # primary results @ nodes
s.print_r() # print reactions @ fixed nodes
s.print_e() # secondary results @ elements

# 7) write file for ParaView
s.write_vtu('bhatti1')
```

It is important to draw the mesh in order to check whether the tags are correct or not – see figure.



The deformed mesh, obtained with ParaView and magnified by 500x, is shown to the right. The numerical results are given below, including the displacements of nodes (`ux` and `uy`) and force reactions (`Rux` and `Ruy`). The axial strain `ea`, axial stress `sa` and axial force `N` are shown to the right.



node	ux	uy
0	0	0
1	0.000538954	-0.000953061
2	0.000264704	-0.000264704
3	0	0

node	Rux	Ruy
0	0.0549267	0.159927
3	-0.0549267	-0.00992667
sum=	1.38778e-17	0.15

elem	ea	sa	N
0	-0.000174295	-34.8591	-0.139436
1	-3.14997e-05	-6.29994	-0.0251998
2	-5.29407e-05	-10.5881	-0.0317644
3	-5.29407e-05	-10.5881	-0.0317644
4	0.000320869	22.4608	0.0449217

Plane Frames ⇒ `EelasticBeam`

Plane Frames ⇒ `EelasticBeam`

Structural members where joints have resistance against rotation are known as beams. The **Euler-Bernoulli** beam theory is a somewhat popular choice for this problem. In this theory, cross sections remain plane and perpendicular during deformations. The corresponding governing equation of motion is:

$$\rho\, A \frac{\partial^2 u_t}{\partial t^2} - \rho\, I \frac{\partial^2 u_t}{\partial t \partial x} + E\, I \frac{\partial^4 u_t}{\partial x^4} = q(x,t)$$

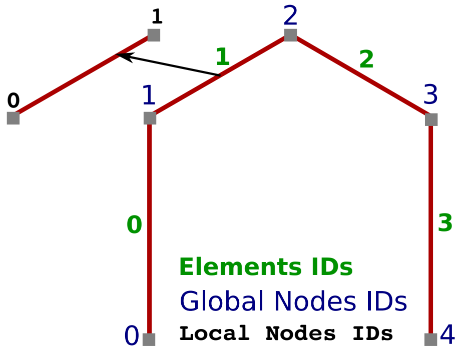
where ρ is the material density, A is the cross-section area, I is the second moment of area about and axis out-of-plane, E is the Young modulus, u_t is the **transverse deflection** of the beam, and q represents a distributed transverse load. These properties are assumed constant along the beam.

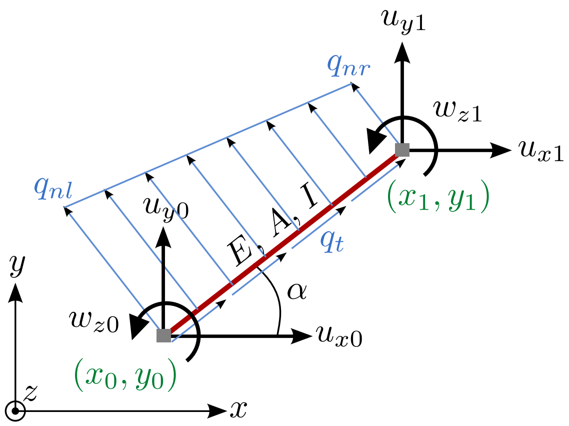
Note that this equation includes a fourth order derivative in space; therefore four boundary conditions must be specified. The FEM can easily handle this and can be be applied employing the same procedure by means of weighted residuals. The Galerkin method can then be applied to the weak-form to obtain a *local* system.

Units
FEM Assembly
Plane Trusses
Plane Frames
EelasticBeam 1
▷ EelasticBeam 2
EelasticBeam 3
EelasticBeam 4
EelasticBeam 5
EelasticBeam 6
EelasticBeam 7
Example 1 a
Example 1 b
Example 1 c
Example 1 d
Example 1 e
2D Diffusion Tri
Stress/Strain
2D Elastic Tri
Summary

After the local system of equations is obtained, the superposition of (space-discretised) equations from rods and beams can be rotated to obtain the *element* equations for *planar frames*. In summary:

- ☐ Each element corresponds to an *Euler-Bernoulli beam* of linear elastic material; these resist axial forces and rotations and can have transversal loads
- ☐ There are 3 solution variables/degrees of freedom per node: u_x and u_y (displacements) and w_z (rotation); hence each element has 6 nodal variables





Units
FEM Assembly
Plane Trusses
Plane Frames
EelasticBeam 1
EelasticBeam 2
▷ EelasticBeam 3
EelasticBeam 4
EelasticBeam 5
EelasticBeam 6
EelasticBeam 7
Example 1 a
Example 1 b
Example 1 c
Example 1 d
Example 1 e
2D Diffusion Tri
Stress/Strain
2D Elastic Tri
Summary

After the FEM space-discretisation based on these assumptions, the full dynamics element equations become

$$\boldsymbol{M}^e \ddot{\boldsymbol{U}}^e + \boldsymbol{K}^e \boldsymbol{U}^e = \boldsymbol{F}^e$$

with

$$\boldsymbol{U}^e = \begin{Bmatrix} u_{x0} \\ u_{y0} \\ w_{z0} \\ u_{x1} \\ u_{y1} \\ w_{z1} \end{Bmatrix} \quad \ddot{\boldsymbol{U}}^e = \begin{Bmatrix} \ddot{u}_{x0} \\ \ddot{u}_{y0} \\ \ddot{w}_{z0} \\ \ddot{u}_{x1} \\ \ddot{u}_{y1} \\ \ddot{w}_{z1} \end{Bmatrix}$$

The static version is the particular case:

$$\boldsymbol{K}^e \boldsymbol{U}^e = \boldsymbol{F}^e$$

It can be shown that:

$$\boldsymbol{K}^e = \boldsymbol{T}^T \boldsymbol{K}^e_l \boldsymbol{T}$$
$$\boldsymbol{M}^e = \boldsymbol{T}^T \boldsymbol{M}^e_l \boldsymbol{T}$$

with:

$$\boldsymbol{T} = \begin{bmatrix} c & s & & & & \\ -s & c & & & & \\ & & 1 & & & \\ & & & c & s & \\ & & & -s & c & \\ & & & & & 1 \end{bmatrix}$$

where $c = \cos\alpha$ and $s = \sin\alpha$ and can be calculated as follows:

$$l = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$$

$$c = \frac{x_1 - x_0}{l} \quad \text{and} \quad s = \frac{y_1 - y_0}{l}$$

Plane Frames ⇒ `EelasticBeam`: element equations

Units
FEM Assembly
Plane Trusses
Plane Frames
EelasticBeam 1
EelasticBeam 2
EelasticBeam 3
▷ EelasticBeam 4
EelasticBeam 5
EelasticBeam 6
EelasticBeam 7
Example 1 a
Example 1 b
Example 1 c
Example 1 d
Example 1 e
2D Diffusion Tri
Stress/Strain
2D Elastic Tri
Summary

The local matrices are:

$$\boldsymbol{K}_l^e = \begin{bmatrix} m & & & -m & & \\ & 12\,n & 6\,l\,n & & -12\,n & 6\,l\,n \\ & 6\,l\,n & 4\,l^2\,n & & -6\,l\,n & 2\,l^2\,n \\ -m & & & m & & \\ & -12\,n & -6\,l\,n & & 12\,n & -6\,l\,n \\ & 6\,l\,n & 2\,l^2\,n & & -6\,l\,n & 4\,l^2\,n \end{bmatrix}$$

$$\boldsymbol{M}_l^e = \frac{\rho\,A\,l}{420}\begin{bmatrix} 140 & & & 70 & & \\ & 156 & 22\,l & & 54 & -13\,l \\ & 22\,l & 4\,l^2 & & 13\,l & -3\,l^2 \\ 70 & & & 140 & & \\ & 54 & 13\,l & & 156 & -22\,l \\ & -13\,l & -3\,l^2 & & -22\,l & 4\,l^2 \end{bmatrix}$$

where $m = \frac{E\,A}{l}$, $n = \frac{E\,I}{l^3}$, E is the Young's modulus, A is the cross-sectional area, I is the cross-sectional area moment of inertia about z (an axis perpendicular to x and y) and l is the beam length.

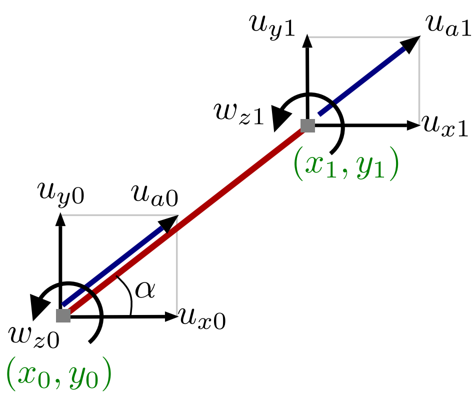
Plane Frames ⇒ `EelasticBeam`: post-computations

Units
FEM Assembly
Plane Trusses
Plane Frames
EelasticBeam 1
EelasticBeam 2
EelasticBeam 3
EelasticBeam 4
▷ EelasticBeam 5
EelasticBeam 6
EelasticBeam 7
Example 1 a
Example 1 b
Example 1 c
Example 1 d
Example 1 e
2D Diffusion Tri
Stress/Strain
2D Elastic Tri
Summary

For plane frames, the following quantities are of great interest, especially when designing structures: axial force N, shear force V and bending moment M.

To calculate these values, **after** the primary values $\boldsymbol{U}^e$ have been found, the displacements in local coordinates have to be computed first:

$$\boldsymbol{U}_l^e = \boldsymbol{T}\boldsymbol{U}^e$$



Plane Frames ⇒ EelasticBeam: Python class

Units
FEM Assembly
Plane Trusses
Plane Frames
EelasticBeam 1
EelasticBeam 2
EelasticBeam 3
EelasticBeam 4
EelasticBeam 5
▷ EelasticBeam 6
EelasticBeam 7
Example 1 a
Example 1 b
Example 1 c
Example 1 d
Example 1 e
2D Diffusion Tri
Stress/Strain
2D Elastic Tri
Summary

The information provided by `EelasticBeam` is:

```python
def info():
    ndim = 2 # number of space dimensions
    nsov = 3 # number of solution variables (2 => ux, uy, wz)

    # vertex boundary conditions info
    vbcs = {'ux':(0,'utype'), # x-displacement
            'uy':(1,'utype'), # y-displacement
            'wz':(2,'utype'), # rotation around z
            'fx':(0,'ftype'), # x-force
            'fy':(1,'ftype'), # y-force
            'mz':(2,'ftype')} # momentum around z

    # edge boundary conditions info
    ebcs  = {}

    # secondary variables
    secvs = ['N', 'V', 'M'] # axial force, shear force, beding moment
    secgs = {'N':'N', 'V':'V', 'M':'M'} # secondary variables groups

    return ndim, nsov, vbcs, ebcs, secvs, secgs
```


Plane Frames ⇒ EelasticBeam: Python class

Units
FEM Assembly
Plane Trusses
Plane Frames
EelasticBeam 1
EelasticBeam 2
EelasticBeam 3
EelasticBeam 4
EelasticBeam 5
EelasticBeam 6
▷ EelasticBeam 7
Example 1 a
Example 1 b
Example 1 c
Example 1 d
Example 1 e
2D Diffusion Tri
Stress/Strain
2D Elastic Tri
Summary

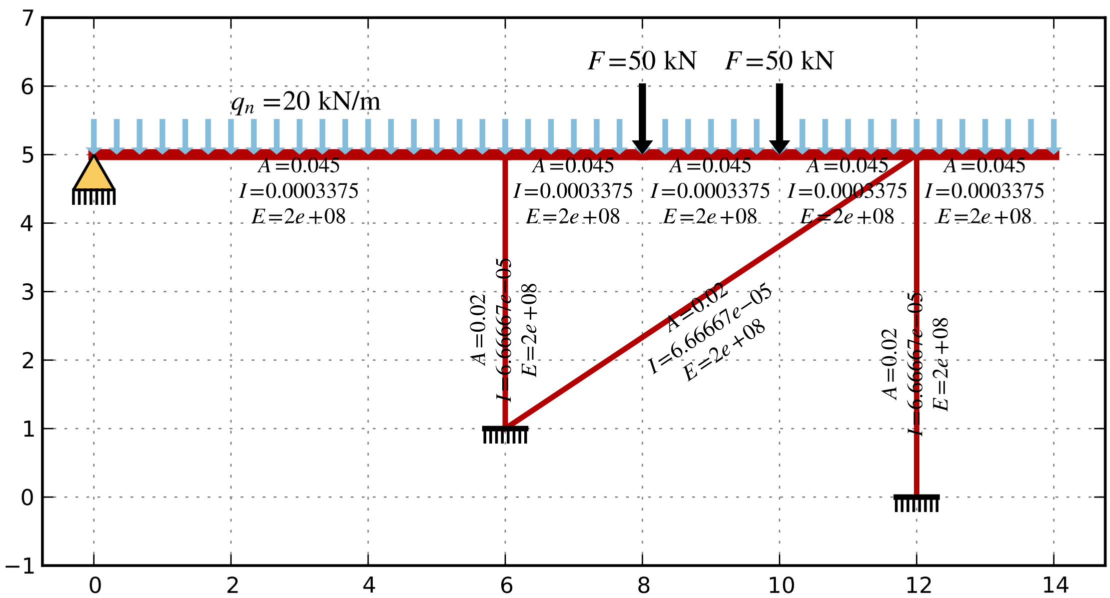
And its constructor requires:

```python
class EelasticBeam:
    def __init__(self, verts, params):
        """
        Elastic beam problem (Euler/Bernoulli)
        ======================================
            Example of input:
                      global_id  tag   x     y
               verts = [[3,    -100, 0.75, 0.0],
                        [4,    -100, 1.0,  0.0]]
              params = {'E':1.0, 'A':1.0, 'I':1.0, 'rho':1.0,
                        'qnqt':(qnl,qnr,qt), 'nop':11}
              Notes: [optional]
                  qnqt is a distributed load along the Beam
                  qnl  is the left value of this load and
                  qnr  is the right value
                  qt   is a tangential distributed load
                  nop  number of points for output of N, V, M
        """
```

Plane Frames ⇒ `EelasticBeam`: example 1

Based on example in [Smith & Griffiths 2006; p130]

For the plane frame below, find the nodal displacements and bending moments at equilibrium state. **Units:** set # 1 in Tab. 1 ⇒ length in m, A in m^2, E in kPa, and I in m^4.



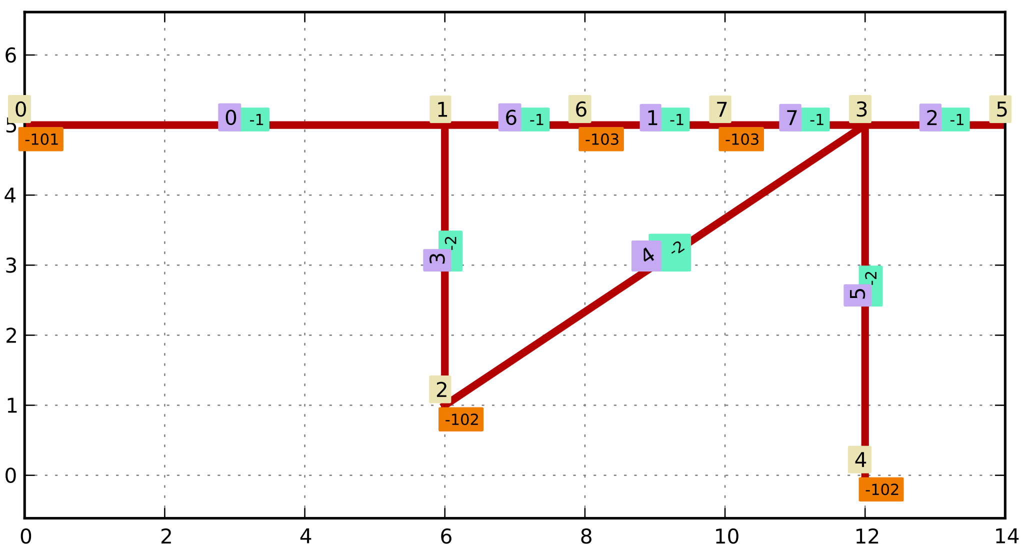
Plane Frames ⇒ `EelasticBeam`: example 1

To solve this problem, the following Python script can be used:

```python
from FEMsolver import *

# 1) input mesh
#     id    tag      x     y
V = [[0,  -101,   0.0,  5.0],
     [1,     0,   6.0,  5.0],
     [2,  -102,   6.0,  1.0],
     [3,     0,  12.0,  5.0],
     [4,  -102,  12.0,  0.0],
     [5,     0,  14.0,  5.0],
     [6,  -103,   8.0,  5.0],
     [7,  -103,  10.0,  5.0]]
#     id   tag   verts
C = [[0,   -1,  [0,1]],
     [1,   -1,  [6,7]],
     [2,   -1,  [3,5]],
     [3,   -2,  [2,1]],
     [4,   -2,  [2,3]],
     [5,   -2,  [4,3]],
     [6,   -1,  [1,6]],
     [7,   -1,  [7,3]]]
m = FEMmesh(V, C)
```

```python
# 2) create dictionary with all parameters
p = {-1:{'E':2.0e8, 'A':4.5e-2, 'I':3.375e-4, # E: [kPa], A: [m2]
         'qnqt':(-20.0,-20.0,0.)},            # I: [m4], qn: [kN/m]
     -2:{'E':2.0e8, 'A':2.0e-2, 'I':(2./3.)*1.0e-4}}

# 3) allocate fem solver object
s = FEMsolver(m, 'EelasticBeam', p)

# 4) set boundary conditions
vb = {-101:{'ux':0.0,'uy':0.0},
      -102:{'ux':0.0,'uy':0.0,'wz':0.0},
      -103:{'fy':-50.0}}  # fy: [kN]
s.set_bcs(vb=vb)

# 5) solve equilibrium problem
s.solve_steady(True) # True ==> with reactions

# 6) print results
s.print_u();  s.print_e()

# 7) plot bending moments
s.beam_plot();  m.show()
```

The generated mesh is:



The output is:

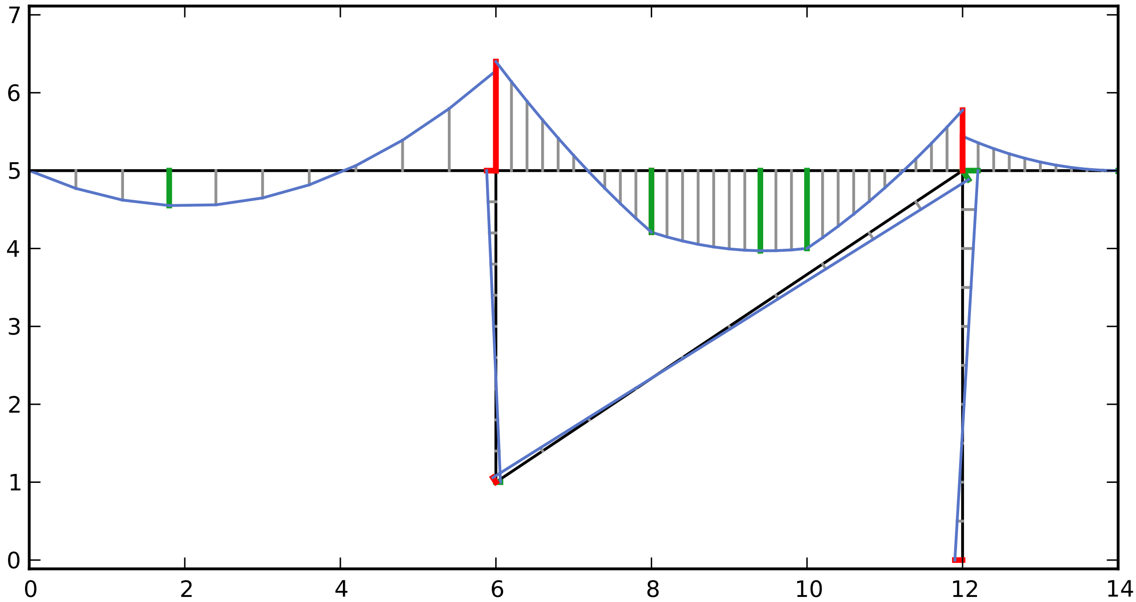
node	ux	uy	wz
0	0	0	-0.000976609
1	1.61917e-05	-0.000198825	-0.000812861
2	0	0	0
3	2.97009e-05	-0.00015039	0.00166799
4	0	0	0
5	2.97009e-05	0.002593	0.00127293
6	2.06948e-05	-0.00342792	-0.00143412
7	2.51978e-05	-0.00378466	0.00117061

node	Rux	Ruy	Rwz
2	29.6631	219.074	1.04798
4	-5.37559	120.312	8.991
0	-24.2876	40.6141	
sum=	-2.84217e-14	380	10.039

elem	N_min	N_max	V_min	V_max	M_min	M_max
0	24.2876	24.2876	-79.3859	40.6141	-116.316	40.7053
1	20.2637	20.2637	-10.5614	29.4386	71.8043	93.4184
2	1.52466e-14	1.52466e-14	-6.39488e-14	40	-40	-1.77636e-15
3	-198.825	-198.825	-4.02383	-4.02383	-10.7572	5.33811
4	-32.5657	-32.5657	2.62656	2.62656	-6.38609	12.5543
5	-120.312	-120.312	5.37559	5.37559	-8.991	17.887
6	20.2637	20.2637	79.4386	119.439	-127.073	71.8043
7	20.2637	20.2637	-100.561	-60.5614	-70.4413	90.6815

With the displacements and rotations at nodes; min/max axial `N` and shear `V` force components within elements; and min/max bending moments `M` within elements. The reactions at fixed nodes are also shown: ex.: the sum of `Ruy` is equal to the sum of external forces.

And the diagram of bending moments is:



Red lines correspond to negative bending moments and green lines to positive values.

2D Diffusion Triangle ⇒ `EdiffusionTri`

2D Diffusion ⇒ `EdiffusionTri`

The problem of finding the temperature $u = T(x, y)$ over a metallic plate or the total hydraulic head $u = H(x, y)$ within a 2D section of soil can be represented by the diffusion equation in 2D:

$$\rho \frac{\partial u}{\partial t} + \frac{\partial}{\partial x} \underbrace{\left(-k_x \frac{\partial u}{\partial x}\right)}_{w_x} + \frac{\partial}{\partial y} \underbrace{\left(-k_y \frac{\partial u}{\partial y}\right)}_{w_y} = s(x)$$

where $s(x)$ is a **source** term (ex: heat generation/unit volume). To solve this PDE, the following boundary conditions can be specified:

$$\begin{aligned} \text{Known } u \text{ along boundary:} \quad & u = \bar{u} \\ \text{Flux specified along boundary:} \quad & \boldsymbol{w} \bullet \hat{\boldsymbol{n}} = \bar{q} \\ \text{Convection along boundary:} \quad & \boldsymbol{w} \bullet \hat{\boldsymbol{n}} = c_c(u - u_c) \end{aligned}$$

where $\boldsymbol{w} = \begin{Bmatrix} w_x \\ w_y \end{Bmatrix}$, $\hat{\boldsymbol{n}}$ is the unit normal at the boundary and

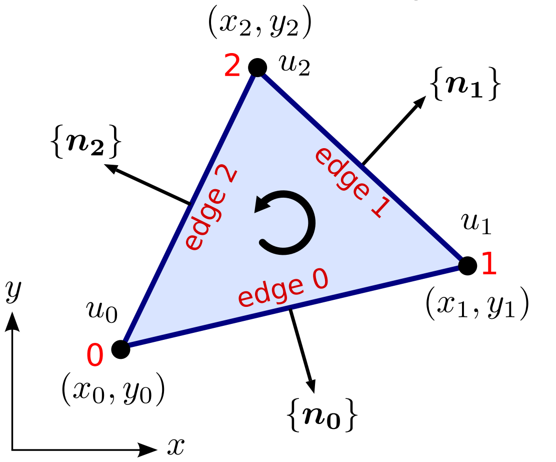
$\boldsymbol{w} \bullet \hat{\boldsymbol{n}} == -k_x \frac{\partial u}{\partial x} n_x - k_y \frac{\partial u}{\partial y} n_y$

2D Diffusion ⇒ `EdiffusionTri`

The following considerations are made:

- ☐ The sign convention for boundary fluxes is: heat/water flowing into a body is positive
- ☐ n_x and n_y are the unit normal components at each side (or edge) of the element

The finite element for this problem is illustrated below:



$$l_{01} = \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$$

$$\boldsymbol{n}_0 = \frac{1}{l_{01}} \begin{Bmatrix} y_1 - y_0 \\ x_0 - x_1 \end{Bmatrix}$$

$$l_{12} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

$$\boldsymbol{n}_1 = \frac{1}{l_{12}} \begin{Bmatrix} y_2 - y_1 \\ x_1 - x_2 \end{Bmatrix}$$

$$l_{20} = \sqrt{(x_2 - x_0)^2 + (y_2 - y_0)^2}$$

$$\boldsymbol{n}_2 = \frac{1}{l_{20}} \begin{Bmatrix} y_0 - y_2 \\ x_2 - x_0 \end{Bmatrix}$$

2D Diffusion ⇒ **`EdiffusionTri`: element equations**

Units
FEM Assembly
Plane Trusses
Plane Frames
2D Diffusion Tri
EdiffusionTri1
EdiffusionTri2
▷ EdiffusionTri3
EdiffusionTri4
EdiffusionTri5
EdiffusionTri6
EdiffusionTri7
EdiffusionTri8
EdiffusionTri9
Example 1 a
Example 1 b
Example 2 a
Example 2 b
Example 2 c
Example 2 d
Example 2 e
Stress/Strain
2D Elastic Tri
Summary

Note that each node has one *solution variable*; hence:

$$\boldsymbol{U}^e = \begin{Bmatrix} u_0 \\ u_1 \\ u_2 \end{Bmatrix}$$

After the finite element space-discretisation, the element/local system is obtained:

$$\boldsymbol{C}^e \dot{\boldsymbol{U}}^e + \boldsymbol{K}^e \boldsymbol{U}^e = \boldsymbol{F}^e$$

where:

$$\boldsymbol{C}^e = \frac{\rho\, A}{12} \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

and $\boldsymbol{K}^e$ is given by two parts:

$$\boldsymbol{K}^e = \boldsymbol{K}^e_k + \boldsymbol{K}^e_c$$

one ($\boldsymbol{K}^e_k$) corresponding to the Poisson's term of the PDE and another ($\boldsymbol{K}^e_c$) due to the convection at the boundaries of the element. This last one is then split into other three parts; one for each side of the triangle.

2D Diffusion ⇒ **`EdiffusionTri`: element equations**

Units
FEM Assembly
Plane Trusses
Plane Frames
2D Diffusion Tri
EdiffusionTri1
EdiffusionTri2
EdiffusionTri3
▷ EdiffusionTri4
EdiffusionTri5
EdiffusionTri6
EdiffusionTri7
EdiffusionTri8
EdiffusionTri9
Example 1 a
Example 1 b
Example 2 a
Example 2 b
Example 2 c
Example 2 d
Example 2 e
Stress/Strain
2D Elastic Tri
Summary

The $\boldsymbol{K}^e_k$ matrix is:

$$\boldsymbol{K}^e_k = \frac{1}{4\,A} \begin{bmatrix} k_x\, b_0\, b_0 + k_y\, c_0\, c_0 & k_x\, b_0\, b_1 + k_y\, c_0\, c_1 & k_x\, b_0\, b_2 + k_y\, c_0\, c_2 \\ k_x\, b_0\, b_1 + k_y\, c_0\, c_1 & k_x\, b_1\, b_1 + k_y\, c_1\, c_1 & k_x\, b_1\, b_2 + k_y\, c_1\, c_2 \\ k_x\, b_0\, b_2 + k_y\, c_0\, c_2 & k_x\, b_1\, b_2 + k_y\, c_1\, c_2 & k_x\, b_2\, b_2 + k_y\, c_2\, c_2 \end{bmatrix}$$

where A is the area of the triangle and the other coefficients can be determined from the nodal coordinates as follows:

$$\begin{aligned} b_0 &= y_1 - y_2 & b_1 &= y_2 - y_0 & b_2 &= y_0 - y_1 \\ c_0 &= x_2 - x_1 & c_1 &= x_0 - x_2 & c_2 &= x_1 - x_0 \\ f_0 &= x_1\, y_2 - x_2\, y_1 & f_1 &= x_2\, y_0 - x_0\, y_2 & f_2 &= x_0\, y_1 - x_1\, y_0 \\ A &= \tfrac{1}{2}(f_0 + f_1 + f_2) \end{aligned}$$

The $\boldsymbol{K}^e_c$ matrix is given by three parts that can all be zero if no convection is specified or by the addition of the non-zero terms corresponding to edges with specified convection:

$$\boldsymbol{K}^e_c = \boldsymbol{K}^e_{c0} + \boldsymbol{K}^e_{c1} + \boldsymbol{K}^e_{c2}$$

2D Diffusion ⇒ `EdiffusionTri`: element equations

The $\boldsymbol{F}^e$ vector also depends on what boundary conditions are specified and is given by three terms:

$$\boldsymbol{F}^e = \underbrace{\boldsymbol{F}^e_s}_{\text{source}} + \underbrace{\boldsymbol{F}^e_f}_{\text{flux}} + \underbrace{\boldsymbol{F}^e_c}_{\text{convection}}$$

where the first two may be given by another three components each:

$$\text{flux:} \qquad \boldsymbol{F}^e_f = \boldsymbol{F}^e_{f0} + \boldsymbol{F}^e_{f1} + \boldsymbol{F}^e_{f2}$$

$$\text{convection:} \qquad \boldsymbol{F}^e_c = \boldsymbol{F}^e_{c0} + \boldsymbol{F}^e_{c1} + \boldsymbol{F}^e_{c2}$$

The $\boldsymbol{F}^e_s$ vector depends on the function $s(x)$; therefore it has to obtained by a (numerical) integration. If $s(x) = c_s$ is a constant, then:

$$\boldsymbol{F}^e_s = \frac{c_s\,A}{3}\begin{Bmatrix}1\\1\\1\end{Bmatrix}$$

2D Diffusion ⇒ `EdiffusionTri`: boundary terms

Depending on what side of the element has specified flux, the following components are computed:

$$\text{edge 0:} \qquad \boldsymbol{F}^e_{f0} = \frac{q_0\,l_{01}}{2}\begin{Bmatrix}1\\1\\0\end{Bmatrix}$$

$$\text{edge 1:} \qquad \boldsymbol{F}^e_{f1} = \frac{q_1\,l_{12}}{2}\begin{Bmatrix}0\\1\\1\end{Bmatrix}$$

$$\text{edge 2:} \qquad \boldsymbol{F}^e_{f2} = \frac{q_2\,l_{20}}{2}\begin{Bmatrix}1\\0\\1\end{Bmatrix}$$

where q_i is the specified flux value at the i edge. The condition $q_i = 0$ automatically corresponds to an insulated or impermeable boundary condition (if convection is not specified).

2D Diffusion ⇒ `EdiffusionTri`: boundary terms

Likewise, depending on what side of the element has specified convection, the following components are computed:

$$\text{edge 0:}\qquad \boldsymbol{K}^e_{c0} = \frac{c_{c0}\, l_{01}}{6}\begin{bmatrix}2 & 1 & 0\\ 1 & 2 & 0\\ 0 & 0 & 0\end{bmatrix} \qquad \boldsymbol{F}^e_{c0} = \frac{c_{c0}\, u_{c0}\, l_{01}}{2}\begin{Bmatrix}1\\1\\0\end{Bmatrix}$$

$$\text{edge 1:}\qquad \boldsymbol{K}^e_{c1} = \frac{c_{c1}\, l_{12}}{6}\begin{bmatrix}0 & 0 & 0\\ 0 & 2 & 1\\ 0 & 1 & 2\end{bmatrix} \qquad \boldsymbol{F}^e_{c1} = \frac{c_{c1}\, u_{c1}\, l_{12}}{2}\begin{Bmatrix}0\\1\\1\end{Bmatrix}$$

$$\text{edge 2:}\qquad \boldsymbol{K}^e_{c2} = \frac{c_{c2}\, l_{20}}{6}\begin{bmatrix}2 & 0 & 1\\ 0 & 0 & 0\\ 1 & 0 & 2\end{bmatrix} \qquad \boldsymbol{F}^e_{c2} = \frac{c_{c2}\, u_{c2}\, l_{20}}{2}\begin{Bmatrix}1\\0\\1\end{Bmatrix}$$

where c_{ci} corresponds to the convection coefficient at edge i and u_{ci} to the surrounding environment temperature at edge i.

2D Diffusion ⇒ `EdiffusionTri`: post-processing

After the element primary values ($\boldsymbol{U}^e$) are found, some interesting secondary variables can be computed. Usually the seepage or conduction velocity are required. These can be easily computed after finding:

$$\begin{Bmatrix}\frac{\partial u}{\partial x}\\ \frac{\partial u}{\partial y}\end{Bmatrix} = \underbrace{\frac{1}{2\,A}\begin{bmatrix}b_0 & b_1 & b_2\\ c_0 & c_1 & c_2\end{bmatrix}}_{\boldsymbol{G}}\begin{Bmatrix}u_0\\u_1\\u_2\end{Bmatrix}$$

where $\boldsymbol{G}$ is known as *gradient matrix*. The seepage velocities can then be calculated:

$$w_x = -k_x\frac{\partial u}{\partial x} \qquad \text{and} \qquad w_y = -k_y\frac{\partial u}{\partial y}$$

Note that these values can be viewed as evaluated at the **centre** of the element. Because neighbours elements may have different velocities, these values are not continuum over the discretised domain.

2D Diffusion ⇒ `EdiffusionTri`: Python class

The information provided by `EdiffusionTri` is:

```
def info():
    # number of space dimensions
    ndim = 2
    # number of solution variables (1 => u)
    nsov = 1 # per node
    # vertex boundary conditions info
    vbcs = {'u':(0,'utype'), # primary value
            'q':(0,'ftype')} # nodal flux
    # edge boundary conditions info
    ebcs = {'u':( 0,'utype'), # primary val
            'q':(-1,'ftype'), # flux
            'c':(-1,'mixed')} # convection
    # secondary variables
    secvs = ['wx', 'wy']
    # secondary variables groups
    secgs = {'w':('wx','wy')}
    return ndim, nsov, vbcs, ebcs, secvs, secgs
```

and its constructor requires:

```
class EdiffusionTri:
    def __init__(self, verts, params):
        """
        Solving:         2          2
              du        d u        d u
          rho == - kx ==== - ky ==== = s(x)
              dt        d x2       d y2

        Example of input:
              global_id  tag   x    y
           verts = [[3,    -100, 0.0, 0.0],
                    [4,    -100, 1.0, 0.0],
                    [1,    -100, 0.0, 1.0]]
           params = {'rho':1., 'kx':1., 'ky':1.,
                     'source':src_val_or_fcn}

        Note:
           src_val_or_fcn: can be a constant value
                           or a callback function
                           such as:
                           lambda x,y: x+y
        """
```

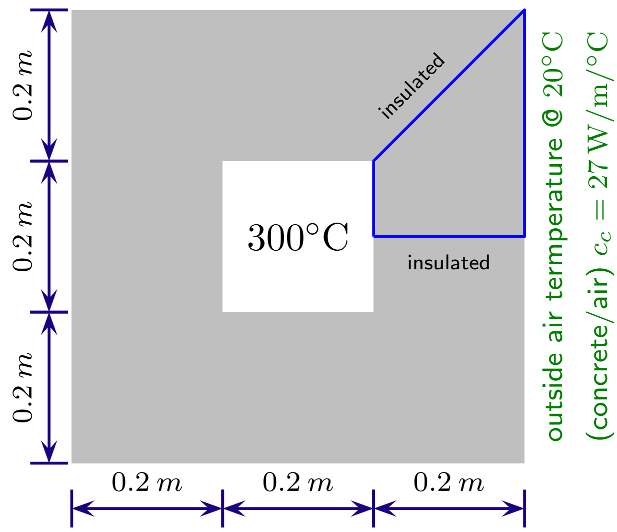

2D Diffusion ⇒ `EdiffusionTri`: example 1

Example from [Bhatti 2005; p28; Example 1.5]

The figure to the right shows a cross section of a chimney where the inner hot gases help to maintain the temperature of the internal surfaces at $300°\mathrm{C}$. This steady state problem can be solved with the diffusion element as follows:

```
from FEMsolver import *
V = [[0, -100, 0.0, 0.0],
     [1,    0, 0.2, 0.0],
     [2,    0, 0.2, 0.3],
     [3, -100, 0.0, 0.1],
     [4,    0, 0.1, 0.1]]
C = [[0, -1, [0,1,4], {}      ],
     [1, -1, [1,2,4], {0:-10}],
     [2, -1, [3,4,2], {}      ],
     [3, -1, [0,4,3], {}      ]]
m = FEMmesh(V, C)
m.draw(); m.show()
```

convection heat transfer coefficient: (concrete/air) $c_c = 27\,\mathrm{W/m/°C}$
outside air termperature @ $20°\mathrm{C}$



Due to symmetry, only one-eighth of the domain needs to be discretised – the highlighted polygon shows the domain considered

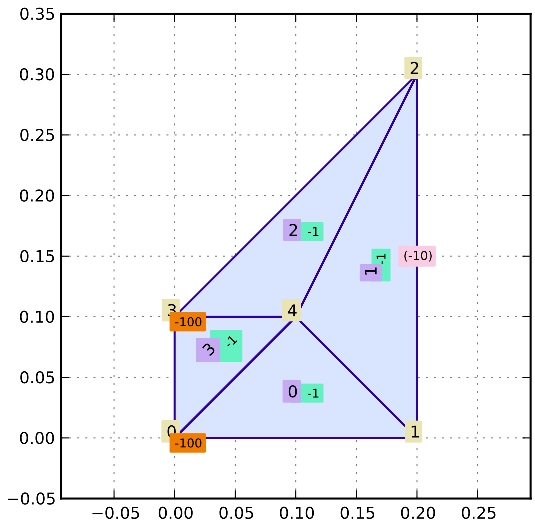
2D Diffusion ⇒ `EdiffusionTri`: example 1

The solution part of the Python code is:

```
# parameters
p = {-1:{'kx':1.4, 'ky':1.4}}
# allocate fem solver object
s = FEMsolver(m, 'EdiffusionTri', p)
# set boundary conditions
vbcs = {-100:{'u':300.0}}
ebcs = {-10:{'c':(27.0, 20.0)}}
s.set_bcs(ebcs, vbcs)
# solve steady
s.solve_steady()
s.print_u(); s.print_e()
```

Note the convection keyword in the `ebcs` dictionary with $c_c = 27\,\mathrm{W/m/°C}$ and $u_c = 20\,°\mathrm{C}$.

The mesh and results are shown to the right with the temperature at nodes `u` and diffusion velocity at elements `wx` and `wy`.



node	u
0	300
1	93.5466
2	23.8437
3	300
4	182.833

elem	wx	wy
0	1445.17	195.168
1	1575.29	325.28
2	1640.34	292.752
3	1640.34	-0

2D Diffusion ⇒ `EdiffusionTri`: example 2

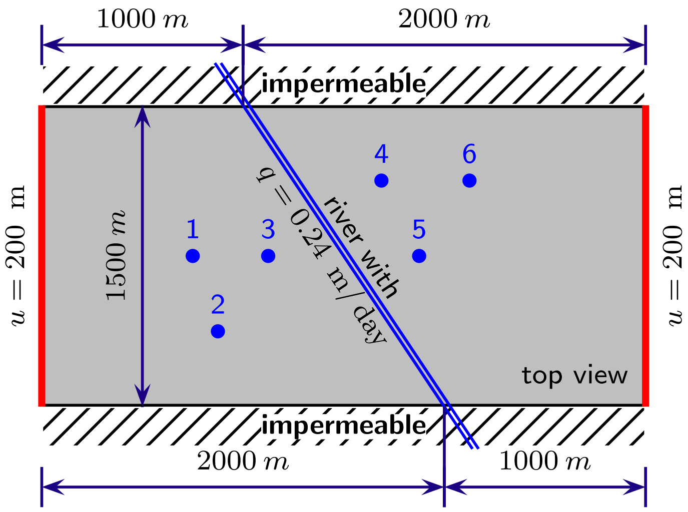
Example from [Reddy 2006; p474; Example 8.5.4]

The figure to the right presents a groundwater aquifer viewed from the top where a river is providing an influx of $0.24\ \mathrm{m/day}$.

The hydraulic (total) head is constant at the left and right boundaries.

The top and bottom boundaries are impermeable (such as in dense rocks).

The blue dots indicate location of pumps.



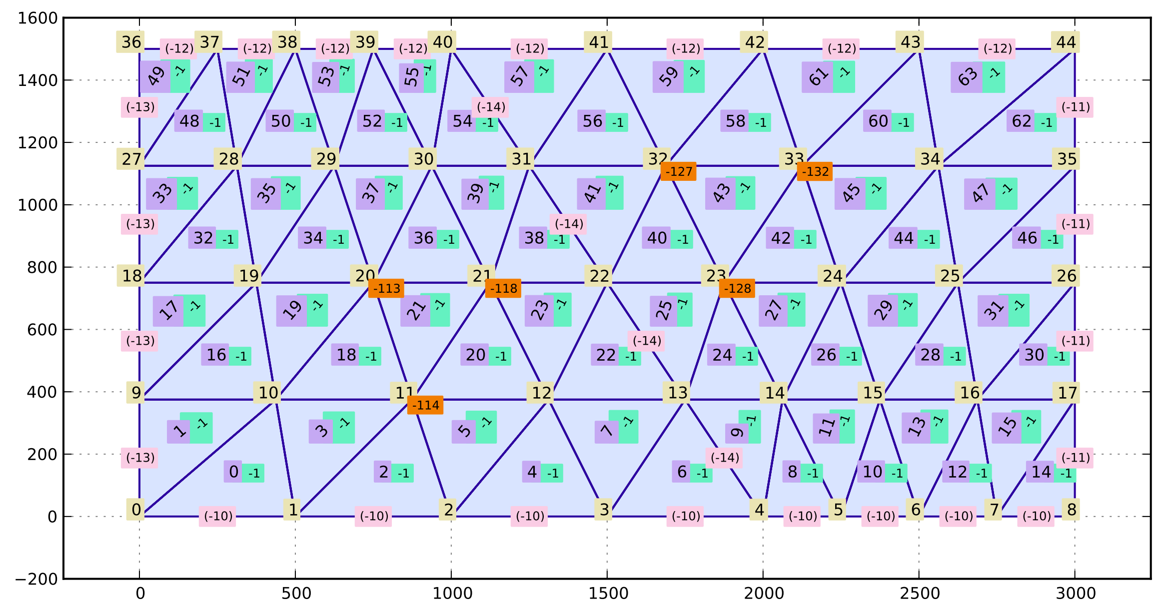
pump 1 @$(750.0,\ 750)$ $q = -229.33$ $\mathrm{m^3/day}$
pump 2 @$(875.0,\ 375)$ $q = -256.0$ $\mathrm{m^3/day}$
pump 3 @$(1125.0,\ 750)$ $q = -714.67$ $\mathrm{m^3/day}$
pump 4 @$(1687.5, 1125)$ $q = -411.429$ $\mathrm{m^3/day}$
pump 5 @$(1875.0,\ 750)$ $q = -1440.0$ $\mathrm{m^3/day}$
pump 6 @$(2125.0, 1125)$ $q = -548.571$ $\mathrm{m^3/day}$

The following code generates the mesh shown in [Reddy 2006]:

```
from FEMsolver import *
# data
Ly        = 1500.0
Lxa, Lxb = 2000.0, 1000.0
nx, ny    = 9, 5
dy        = Ly/float(ny-1)
neh       = (nx-1)/2 # number of
# vertices              #   half of
V = []                  #   columns of
for j in range(ny): #   elements
    dx1 = Lxa/float(neh)
    dx2 = Lxb/float(neh)
    l1, l2, di, swap = 0,0,0, True
    for i in range(nx):
        if i > neh and swap:
            dx1,dx2   = dx2,dx1
            l1,l2,di = Lxa,Lxb,neh
            swap      = False
        x1 = l1 + (i-di)*dx1 # bot
        x2 = l2 + (i-di)*dx2 # top
        y  = j * dy
        x  = x1 + (x2-x1) * y / Ly
        V.append([i+j*nx, 0, x, y])
```

```
C = [] # cells
for je in range(ny-1):      # for each column of elements
    for ie in range(nx-1): # for each row of elements
        tg0 = {} # tags of edges of first triangle
        tg1 = {} # tags of edges of second triangle
        vv0 = [ie+je*nx, ie+je*nx+1,     ie+(je+1)*nx+1]
        vv1 = [ie+je*nx, ie+(je+1)*nx+1, ie+(je+1)*nx  ]
        if je == 0:     tg0[0] = -10    # tag: bottom
        if ie == nx-2:  tg0[1] = -11    # tag: right
        if je == ny-2:  tg1[1] = -12    # tag: top
        if ie == 0:     tg1[2] = -13    # tag: left
        if ie == neh-1: tg0[1] = -14    # tag: river
        C.append([len(C), -1, vv0, tg0]) # first triang
        C.append([len(C), -1, vv1, tg1]) # second trian
# mesh
m = FEMmesh(V, C) # tag vertices:
m.tag_vert(-113,  750.0,  750.)
m.tag_vert(-114,  875.0,  375.)
m.tag_vert(-118, 1125.0,  750.)
m.tag_vert(-127, 1687.5, 1125.)
m.tag_vert(-128, 1875.0,  750.)
m.tag_vert(-132, 2125.0, 1125.)
m.draw();  m.show()
```


The generated mesh is shown below:



Units
FEM Assembly
Plane Trusses
Plane Frames
2D Diffusion Tri
EdiffusionTri1
EdiffusionTri2
EdiffusionTri3
EdiffusionTri4
EdiffusionTri5
EdiffusionTri6
EdiffusionTri7
EdiffusionTri8
EdiffusionTri9
Example 1 a
Example 1 b
Example 2 a
Example 2 b
Example 2 c
▷ Example 2 d
Example 2 e
Stress/Strain
2D Elastic Tri
Summary

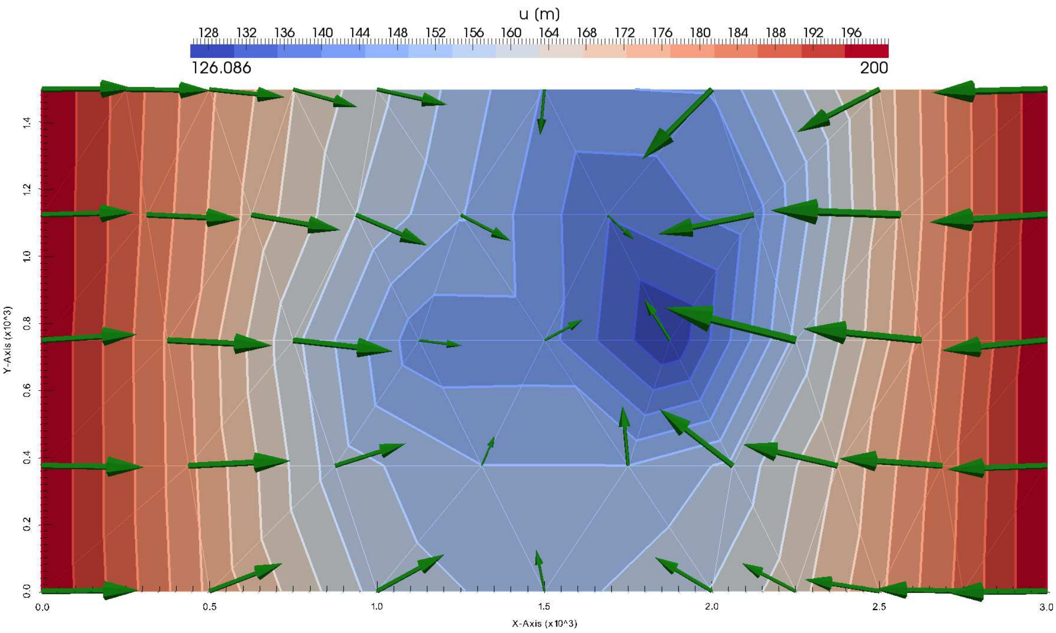
To solve this problem the script shown to the right can be used.

The command `write_vtu` writes an output file for visualisation in ParaView. Note that to get the secondary variables w_x and w_y (which are collected in a vector $\boldsymbol{w}$) an extrapolation has to be made from values at the centre of elements to the surrounding nodes. This can be accomplished with the `extrapolate` command.

```
from FEMsolver import *
# import mesh here
from aquifer_mesh import *
# parameters
p = {-1:{'kx':20.0, 'ky':20.0}}
# solver
s = FEMsolver(m,'EdiffusionTri',p)
# boundary conditions
vb = {-113:{'q': -229.33 }, # nodal flux
      -114:{'q': -256.0  },
      -118:{'q': -714.67 },
      -127:{'q': -411.429},
      -128:{'q':-1440.0  },
      -132:{'q': -548.571}}
eb = { -11:{'u':  200.0  }, # hydr head
       -13:{'u':  200.0  }, #   on edge
       -14:{'q':    0.24 }} # edge flux
s.set_bcs(vb=vb, eb=eb)
# solve
s.solve_steady()
# write file for ParaView
s.extrapolate()
s.write_vtu('aquifer')
```


Units
FEM Assembly
Plane Trusses
Plane Frames
2D Diffusion Tri
EdiffusionTri1
EdiffusionTri2
EdiffusionTri3
EdiffusionTri4
EdiffusionTri5
EdiffusionTri6
EdiffusionTri7
EdiffusionTri8
EdiffusionTri9
Example 1 a
Example 1 b
Example 2 a
Example 2 b
Example 2 c
Example 2 d
▷ Example 2 e
Stress/Strain
2D Elastic Tri
Summary

Using ParaView, the following figure is obtained showing the contour of equal hydraulic heads and the seepage vector field.



Continuum Mechanics, Elasticity, and Stresses and Strains

Linear elasticity and stresses and strains in 3D

The deformation and equilibrium of continuum bodies can be at times represented by linear elasticity. Of course it depends on whether the material of the domain under study can represented by a linear elastic model or not. Assuming small strains, two quantities a of great importance in elasticity are the *Cauchy* stress tensor:

$$\underset{\approx}{\boldsymbol{\sigma}}_{x-y-z} = \boldsymbol{\sigma} = \begin{bmatrix} \sigma_x & \sigma_{xy} & \sigma_{zx} \\ \sigma_{xy} & \sigma_y & \sigma_{yz} \\ \sigma_{zx} & \sigma_{yz} & \sigma_z \end{bmatrix}$$

and the small strains tensor (linear strain):

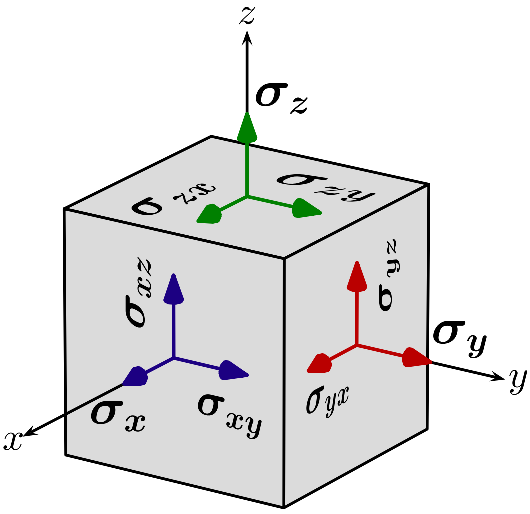
$$\underset{\approx}{\boldsymbol{\varepsilon}}_{x-y-z} = \boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_x & \varepsilon_{xy} & \varepsilon_{zx} \\ \varepsilon_{xy} & \varepsilon_y & \varepsilon_{yz} \\ \varepsilon_{zx} & \varepsilon_{yz} & \varepsilon_z \end{bmatrix}$$

where all components are referred to an ortho-normal Cartesian system $x - y - z$; the laboratory system of reference.

In *Solid Mechanics*, the sign convention for stress components is such that *tensile stresses are positive*. Although this is just a convention, thinking of tensile stresses as the ones causing *stretching* if fairly natural.

Sometimes in *Soil Mechanics* the convention is opposite to this one.

The stress state in a point is usually illustrated with the aid of an infinitesimal cube as in the figure to the right. In this figure, σ_{ij} (and ε_{ij}) are the **components** of the stress and strain tensors, respectively, with respect to the laboratory reference system; usually a Cartesian ortho-normal one.



$$\underset{\sim}{\boldsymbol{\sigma}}_{lab} = \begin{bmatrix} \sigma_x & \sigma_{xy} & \sigma_{zx} \\ \sigma_{xy} & \sigma_y & \sigma_{yz} \\ \sigma_{zx} & \sigma_{yz} & \sigma_z \end{bmatrix}$$

Tensors in 3D can be also represented as vectors in a 9D system. Because of symmetry, some components can be dropped and a 6D system can be used. Actually, the way we organise (visualise) the components of tensors is fairly arbitrary. To simplify computations, the so-called Mandel's basis is selected leading to the following notation:

$$\underset{\sim}{\boldsymbol{\sigma}}_{x-y-z} = \boldsymbol{\sigma} = \begin{Bmatrix} \sigma_x \\ \sigma_y \\ \sigma_z \\ \sqrt{2}\,\sigma_{xy} \\ \sqrt{2}\,\sigma_{yz} \\ \sqrt{2}\,\sigma_{zx} \end{Bmatrix} \qquad \underset{\sim}{\boldsymbol{\varepsilon}}_{x-y-z} = \boldsymbol{\varepsilon} = \begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \varepsilon_z \\ \sqrt{2}\,\varepsilon_{xy} \\ \sqrt{2}\,\varepsilon_{yz} \\ \sqrt{2}\,\varepsilon_{zx} \end{Bmatrix}$$

Note that the $\sqrt{2}$ coefficient guarantees that the mapping is isomorphic; hence:

$$\|\underset{\sim}{\boldsymbol{\sigma}}\| = \sum_{i=x,y,z} \sum_{j=x,y,z} \sqrt{\sigma_{ij}\sigma_{ij}} = \sqrt{\boldsymbol{\sigma}^T \boldsymbol{\sigma}}$$

The theory of Elasticity provides a mathematical model for elastic materials by relating stresses with strains. This type of model is also known as *constitutive* since it represents a material (mechanical) behaviour. In 3D, the relationship is:

$$\begin{Bmatrix} \sigma_x \\ \sigma_y \\ \sigma_z \\ \sqrt{2}\,\sigma_{xy} \\ \sqrt{2}\,\sigma_{yz} \\ \sqrt{2}\,\sigma_{zx} \end{Bmatrix} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & 1-2\nu & 0 & 0 \\ 0 & 0 & 0 & 0 & 1-2\nu & 0 \\ 0 & 0 & 0 & 0 & 0 & 1-2\nu \end{bmatrix} \begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \varepsilon_z \\ \sqrt{2}\,\varepsilon_{xy} \\ \sqrt{2}\,\varepsilon_{yz} \\ \sqrt{2}\,\varepsilon_{zx} \end{Bmatrix}$$

where E is the Young's modulus and ν is the Poisson coefficient.

Note that there is also a number of other elastic constants, such as K (bulk modulus) and G (shear modulus) that can be directly calculated from E and ν (and vice-versa). Also note that **Elasticity most always does not apply to soils**, since these materials have highly non-linear mechanical behaviour. Nonetheless, Elasticity is a useful starting point to understand stresses and strains distributions.

The concept of *invariants* can be illustrated with the case of a 3D vector in a $x - y - z$ Cartesian system of reference. Although the components of this vector change when we choose a different coordinate system (say, cylindrical), its magnitude never changes; i.e. it is *invariant* (w.r.t observer).

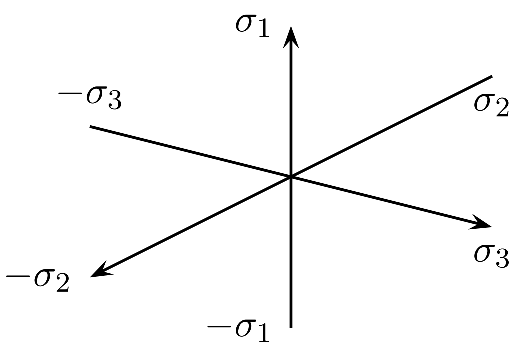
Second order tensors such as stress and strain have also invariant quantities; actually they have 3 independent ones – also known as *characteristic* invariants I_i:

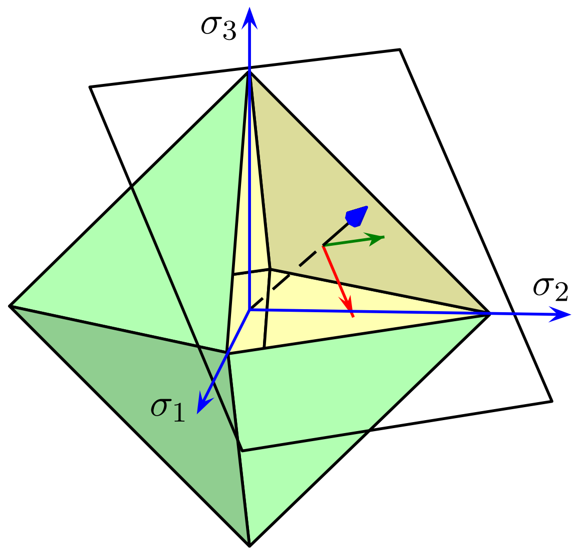
$$I_1 = \operatorname{tr} \boldsymbol{a}$$
$$I_2 = \frac{1}{2}\left(\operatorname{tr} \boldsymbol{a}\right)^2 - \frac{1}{2}\operatorname{tr}\left(\boldsymbol{a} \bullet \boldsymbol{a}\right)$$
$$I_3 = \det \boldsymbol{a}$$

Any combination of I_i will also be invariant. Therefore, other quantities with better physical meaning are derived. Four important ones; two from stresses and two from strains, are commonly used in material modeling using continuum mechanics: p_{cam}, q_{cam}, ε_v, and ε_d.

Haigh-Westergaard and Octahedral diagrams

A very convenient diagram to plot the principal stress components of a stress state at a given point is the *Haigh-Westergaard* principal stress space. In this space, eight (virtual) planes can be drawn for the stress state, allowing the visualisation and definition of *stress invariants*.



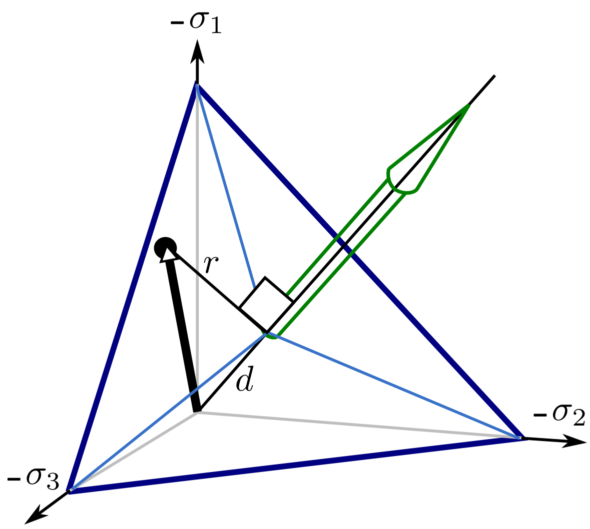


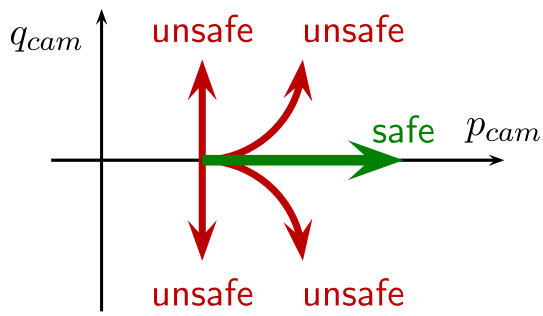
Cambridge Stress Invariants

With the aid of one octahedral plane, two geometric entities can be defined: the radius r and the distance from the origin d, corresponding to a stress state triple $(-\sigma_1,-\sigma_2,-\sigma_3)$. Then, two important stress invariants are defined:

$$p_{cam} = \frac{d}{\sqrt{3}} \quad \text{and} \quad q_{cam} = \sqrt{\frac{3}{2}}\, r$$

where p_{cam} is known as the Cambridge *mean* stress invariant and q_{cam} as the *deviatoric* stress invariant. Each one has a more or less physical meaning related to the proximity of failure of a stress point.





Volume change

Regarding the strain tensor, another set of useful invariants includes the *volumetric* and *deviatoric* strains: ε_v and ε_d, respectively. To obtain the first one, the volume change in an infinitesimal cube can be considered as follows:

$$\begin{aligned}\delta V &= V_f - V_0 \\ &= (L_x + \delta u_x)(L_y + \delta u_y)(L_z + \delta u_z) - L_x\, L_y\, L_z \\ &= \delta u_x\, L_y\, L_z + \delta u_y\, L_z\, L_x + \delta u_z\, L_x\, L_y + \\ &\qquad \delta u_x\, \delta u_y\, L_z + \delta u_y\, \delta u_z\, L_x + \delta u_z\, \delta u_x\, L_y + \\ &\qquad \delta u_x\, \delta u_y\, \delta u_z\end{aligned}$$

Now, assuming small changes (small strains):

$$\delta u_x\, \delta u_y = \delta u_y\, \delta u_z = \delta u_z\, \delta u_x = \delta u_x\, \delta u_y\, \delta u_z \approx 0$$

Then:

$$\delta V = \delta u_x\, L_y\, L_z + \delta u_y\, L_z\, L_x + \delta u_z\, L_x\, L_y$$

Strain Invariants: Volumetric Strain

Then, the volumetric strain can be obtained (expansion is positive):

$$\begin{aligned}\varepsilon_v &= \varepsilon_x + \varepsilon_y + \varepsilon_z \\ &= \frac{\delta u_x}{L_x} + \frac{\delta u_y}{L_y} + \frac{\delta u_z}{L_z} \\ &= \frac{\delta u_x\, L_y\, L_z + \delta u_y\, L_z\, L_x + \delta u_z\, L_x\, L_y}{L_x\, L_y\, L_z} \\ &= \frac{\delta V}{V}\end{aligned}$$

This can be related with the *void ratio* e usually discussed in Soil Mechanics books:

$$\varepsilon_v = \frac{\Delta e}{1+e}$$

The deviatoric strain invariant can be calculated as follows:

$$\varepsilon_d = \frac{\sqrt{2}}{3}\sqrt{(\varepsilon_1 - \varepsilon_2)^2 + (\varepsilon_2 - \varepsilon_3)^2 + (\varepsilon_3 - \varepsilon_1)^2}$$

and is a type of average of the differences between all principal strains ε_i

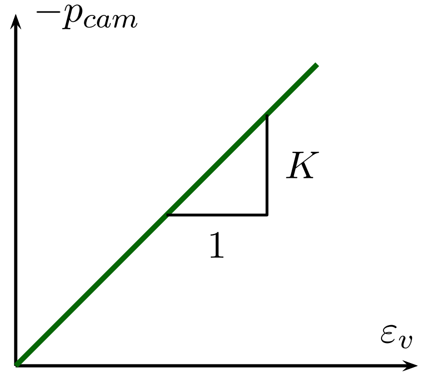
Elasticity with Invariants

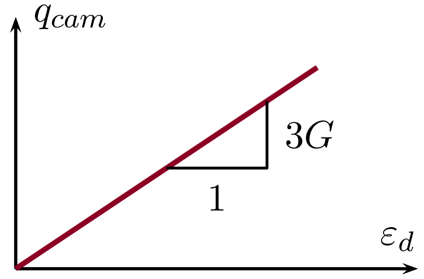
Apart from the better physical meaning, an advantage of using stress and strain invariants is when developing or calibrating models for the mechanical behaviour of materials. For instance, an linear elastic model can be expressed as illustrated to the right, where K is the *bulk modulus* and G is the *shear modulus*. These can be calculated as follows:

$$K = \frac{E}{3\,(1 - 2\,\nu)} \qquad G = \frac{E}{2\,(1 + \nu)}$$

And vice-versa, the Young's modulus and Poisson's coefficient can be calculated by:

$$E = \frac{9\,K\,G}{3K + G} \qquad \nu = \frac{3K - 2G}{6K + 2G}$$





Effective Stresses in Soil Mechanics

In Soil Mechanics, a fundamental concept is the one related to the porous structure of soils. In this theory, water can be present within the soil matrix and even sustain all external pressure. It is assumed that part of the stresses goes to the solid skeleton and part to the interstitial water (**pore-water**).

With water in its voids, the mechanical behaviour of soils has to be modelled by introducing a measure of *effective stress* ($\underset{\approx}{\boldsymbol{\sigma}}'$). In standard literature, this is accomplished with:

$$\underset{\approx}{\boldsymbol{\sigma}}' = \underset{\approx}{\boldsymbol{\sigma}} + p_w\,\underset{\approx}{\boldsymbol{I}}$$

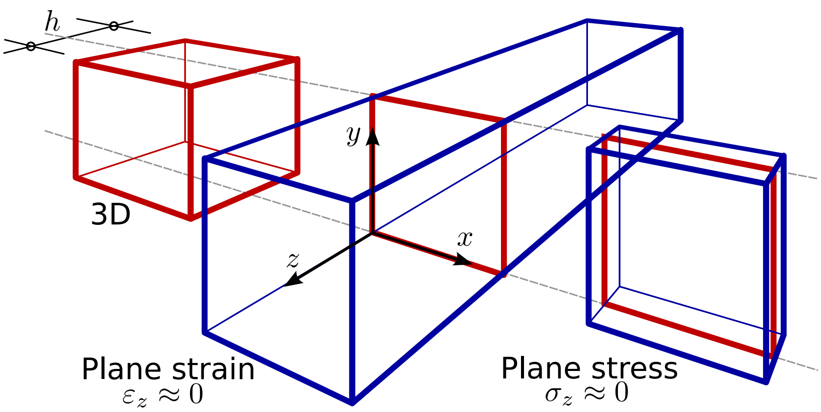
where p_w is the pore-water pressure and $\underset{\approx}{\boldsymbol{I}}$ is the second-order identity tensor. In components form:

$$\begin{bmatrix} \sigma'_x & \sigma'_{xy} & \sigma'_{xz} \\ \sigma'_{yx} & \sigma'_y & \sigma'_{yz} \\ \sigma'_{zx} & \sigma'_{zy} & \sigma'_z \end{bmatrix} = \begin{bmatrix} \sigma_x & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_y & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_z \end{bmatrix} + p_w \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Note that we have adopted the **Solid** mechanics sign convention.

Many equilibrium problems have to be studied considering three dimensions; however some situations can be conveniently simplified to 2D. Three common cases are the **axis-symmetric**, **plane stress**, and **plane-strain** conditions. The last two are considered here. Some examples are:

- ☐ Plane stress ⇒ beams, brackets, hooks, ...
- ☐ Plane strain ⇒ dams, tunnels, shafts, ...



Employing the mentioned simplifications:

- ☐ For plane stress: $\sigma_z \approx 0$, $\sigma_{yz} \approx 0$, and $\sigma_{zx} \approx 0$, **but:** $\varepsilon_z = ?$

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_x & \sigma_{xy} & 0 \\ \sigma_{xy} & \sigma_y & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \text{and} \quad \boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_x & \varepsilon_{xy} & 0 \\ \varepsilon_{xy} & \varepsilon_y & 0 \\ 0 & 0 & \varepsilon_z \end{bmatrix}$$

- ☐ For plane strain: $\varepsilon_z \approx 0$, $\varepsilon_{yz} \approx 0$, and $\varepsilon_{zx} \approx 0$, **but:** $\sigma_z = ?$

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_x & \sigma_{xy} & 0 \\ \sigma_{xy} & \sigma_y & 0 \\ 0 & 0 & \sigma_z \end{bmatrix} \quad \text{and} \quad \boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_x & \varepsilon_{xy} & 0 \\ \varepsilon_{xy} & \varepsilon_y & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Hence, at most four components are enough in calculations; employing Mandel's basis:

$$\boldsymbol{\sigma} = \begin{Bmatrix} \sigma_x \\ \sigma_y \\ \sigma_z \\ \sqrt{2}\,\sigma_{xy} \end{Bmatrix} \quad \text{and} \quad \boldsymbol{\varepsilon} = \begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \varepsilon_z \\ \sqrt{2}\,\varepsilon_{xy} \end{Bmatrix}$$

2D Elasticity Triangle ⇒ `EelasticTri`

2D Elasticity ⇒ `EelasticTri`: element equations

The system of partial differential equations representing equilibrium problems and/or dynamics of deformable bodies is derived from the **balance of linear and angular momenta**. In small strain linear elasticity, the angular balance law implies that stresses are symmetric. The balance of linear momentum is given by:

$$\mathbf{div}\,\boldsymbol{\sigma} + \rho\,\boldsymbol{b} = \rho\,\boldsymbol{a}$$

or, in 2D:

$$\frac{\partial \sigma_x}{\partial x} + \frac{\partial \sigma_{xy}}{\partial y} + \rho\, b_x = \rho\, a_x$$
$$\frac{\partial \sigma_{xy}}{\partial x} + \frac{\partial \sigma_y}{\partial y} + \rho\, b_y = \rho\, a_y$$

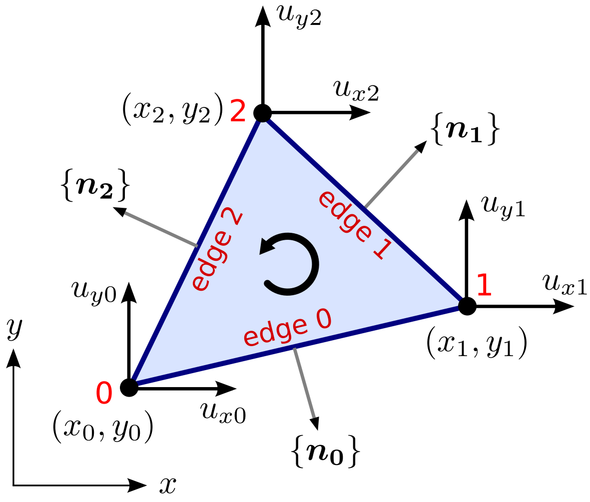
where ρ is the material density, $\boldsymbol{b}$ a vector of body forces, and $\boldsymbol{a}$ a vector of accelerations. Note that the system above is of second order because, in elasticity, $\boldsymbol{\sigma}$ is a function of $\boldsymbol{\varepsilon}$ and:

$$\varepsilon_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \qquad u_i \text{ are the } x, y\text{-displacements}$$

2D Elasticity ⇒ EelasticTri: element equations

The finite element space-discretisation of elasticity problems in 2D using triangles (with 3 nodes) assumes:

- linear displacements over the element and constant strains and stresses within the element ⇒ constant strain element (CST)
- 2 solution variables/degrees of freedom per node (u_x and u_y) ⇒ 6 nodal variables
- the triangle thickness is 1 unit of length for plane-strain problems or a given value of thickness for plane-stress problems



2D Elasticity ⇒ EelasticTri: element equations

The element equations thus obtained are:

$$\boldsymbol{M}^e \ddot{\boldsymbol{U}}^e + \boldsymbol{K}^e \boldsymbol{U}^e = \boldsymbol{F}^e$$

where:

$$\boldsymbol{M}^e = \frac{\rho\, h\, A}{12} \begin{bmatrix} 2 & 0 & 1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 1 & 0 & 1 \\ 1 & 0 & 2 & 0 & 1 & 0 \\ 0 & 1 & 0 & 2 & 0 & 1 \\ 1 & 0 & 1 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 & 0 & 2 \end{bmatrix} \quad \boldsymbol{U}^e = \begin{Bmatrix} u_{x0} \\ u_{y0} \\ u_{x1} \\ u_{y1} \\ u_{x2} \\ u_{y2} \end{Bmatrix} \quad \ddot{\boldsymbol{U}}^e = \begin{Bmatrix} \ddot{u}_{x0} \\ \ddot{u}_{y0} \\ \ddot{u}_{x1} \\ \ddot{u}_{y1} \\ \ddot{u}_{x2} \\ \ddot{u}_{y2} \end{Bmatrix}$$

where ρ is the material density, h is the thickness of the element ($h = 1$ for plane-strain), and A is the area of the element.

$\boldsymbol{F}^e$ depends on the boundary conditions (distributed loads applied to the edges) and $\boldsymbol{K}^e$ is given by:

$$\boldsymbol{K}^e = h\, A\, \boldsymbol{B}^T \boldsymbol{D} \boldsymbol{B}$$

2D Elasticity $\Rightarrow$ `EelasticTri`: element equations

$\boldsymbol{B}$ matrix is known as **strain-displacement matrix** and is given by:

$$\boldsymbol{B} = \frac{1}{2\,A}\begin{bmatrix} b_0 & 0 & b_1 & 0 & b_2 & 0 \\ 0 & c_0 & 0 & c_1 & 0 & c_2 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ c_0/\sqrt{2} & b_0/\sqrt{2} & c_1/\sqrt{2} & b_1/\sqrt{2} & c_2/\sqrt{2} & b_2/\sqrt{2} \end{bmatrix}$$

with:

$$b_0 = y_1 - y_2 \qquad c_0 = x_2 - x_1$$
$$b_1 = y_2 - y_0 \qquad c_1 = x_0 - x_2$$
$$b_2 = y_0 - y_1 \qquad c_2 = x_1 - x_0$$

$$f_0 = x_1\,y_2 - x_2\,y_1$$
$$f_1 = x_2\,y_0 - x_0\,y_2$$
$$f_2 = x_0\,y_1 - x_1\,y_0$$
$$A = \tfrac{1}{2}(f_0 + f_1 + f_2)$$

also:

$$l_{01} = \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$$
$$\boldsymbol{n}_0 = \frac{1}{l_{01}}\begin{Bmatrix} y_1 - y_0 \\ x_0 - x_1 \end{Bmatrix}$$
$$l_{12} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$
$$\boldsymbol{n}_1 = \frac{1}{l_{12}}\begin{Bmatrix} y_2 - y_1 \\ x_1 - x_2 \end{Bmatrix}$$
$$l_{20} = \sqrt{(x_2 - x_0)^2 + (y_2 - y_0)^2}$$
$$\boldsymbol{n}_2 = \frac{1}{l_{20}}\begin{Bmatrix} y_0 - y_2 \\ x_2 - x_0 \end{Bmatrix}$$

2D Elasticity $\Rightarrow$ `EelasticTri`: element equations

$\boldsymbol{D}$ matrix is known as **constitutive matrix** and is given by:

- For **plane stress**:

$$\boldsymbol{D} = \frac{E}{1-\nu^2}\begin{bmatrix} 1 & \nu & 0 & 0 \\ \nu & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1-\nu \end{bmatrix}$$

- For **plane strain**:

$$\boldsymbol{D} = \frac{E}{(1+\nu)(1-2\nu)}\begin{bmatrix} 1-\nu & \nu & \nu & 0 \\ \nu & 1-\nu & \nu & 0 \\ \nu & \nu & 1-\nu & 0 \\ 0 & 0 & 0 & 1-2\nu \end{bmatrix}$$

where E is the Young modulus and ν the Poisson's coefficient.

2D Elasticity ⇒ `EelasticTri`: element equations

The vector $\boldsymbol{F}^e$ is given by three terms:

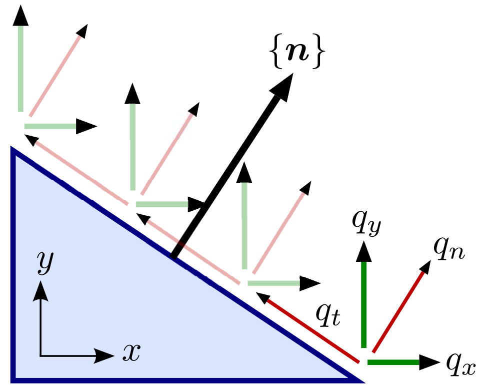
$$\boldsymbol{F}^e = \boldsymbol{F}^e_0 + \boldsymbol{F}^e_1 + \boldsymbol{F}^e_2$$

each corresponding to one edge $i = 0, 1, 2$ of the triangle:

$$\boldsymbol{F}^e_0 = \frac{h\,l_{01}}{2}\begin{Bmatrix} q_x \\ q_y \\ q_x \\ q_y \\ 0 \\ 0 \end{Bmatrix} \qquad \boldsymbol{F}^e_1 = \frac{h\,l_{12}}{2}\begin{Bmatrix} 0 \\ 0 \\ q_x \\ q_y \\ q_x \\ q_y \end{Bmatrix} \qquad \boldsymbol{F}^e_2 = \frac{h\,l_{20}}{2}\begin{Bmatrix} q_x \\ q_y \\ 0 \\ 0 \\ q_x \\ q_y \end{Bmatrix}$$

Either (q_x, q_y) or (q_n, q_t) can be specified because:

$$q_x = n_x\,q_n - n_y\,q_t$$
$$q_y = n_y\,q_n + n_x\,q_t$$



2D Elasticity ⇒ `EelasticTri`: post-processing

After obtaining the element primary values (displacements) $\boldsymbol{U}^e$, the strains (constant) within the element can be found with:

$$\boldsymbol{\varepsilon} = \boldsymbol{B}\,\boldsymbol{U}^e$$

The stresses can then be calculated with:

$$\boldsymbol{\sigma} = \boldsymbol{D}\,\boldsymbol{\varepsilon}$$

Note that, for the plane stress situation, σ_z will always be zero because:

$$\begin{Bmatrix} \sigma_x \\ \sigma_y \\ \sigma_z \\ \sqrt{2}\,\sigma_{xy} \end{Bmatrix} = \frac{E}{1-\nu^2}\begin{bmatrix} 1 & \nu & 0 & 0 \\ \nu & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1-\nu \end{bmatrix}\begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \varepsilon_z \\ \sqrt{2}\,\varepsilon_{xy} \end{Bmatrix} \qquad \text{plane-stress}$$

In this case, the out-of-plane strain ε_z has to be post-computed using the following expression:

$$\varepsilon_z = \frac{-\nu}{E}(\sigma_x + \sigma_y)$$

2D Elasticity ⇒ `EelasticTri`: Python class

Units
FEM Assembly
Plane Trusses
Plane Frames
2D Diffusion Tri
Stress/Strain
2D Elastic Tri
EelasticTri 1
EelasticTri 2
EelasticTri 3
EelasticTri 4
EelasticTri 5
EelasticTri 6
EelasticTri 7
▷ EelasticTri 8
Example 1 a
Example 1 b
Example 2 a
Example 2 b
Example 2 c
Example 2 d
Example 2 e
Example 2 f
Example 2 g
Example 2 h
Summary

The information for `EelasticTri` is:

```python
def info():
    # number of space dimensions
    ndim = 2
    # number of solution variables (2 => ux, uy)
    nsov = 2
    # vertex boundary conditions info
    vbcs = {'ux':(0,'utype'), # x-displacement
            'uy':(1,'utype'), # y-displacement
            'fx':(0,'ftype'), # x-force
            'fy':(1,'ftype')} # y-force
    # edge boundary conditions info
    ebcs = {'ux'  :( 0,'utype'), # x-displacement
            'uy'  :( 1,'utype'), # y-displacement
            'qxqy':(-1,'ftype'), # distr load (local)
            'qnqt':(-1,'ftype')} # distr load (nor/tan)
    # secondary variables (stresses and strains)
    secvs = ['sx', 'sy', 'sz', 'sxy', 'p', 'q',
             'ex', 'ey', 'ez', 'exy', 'ev', 'ed']
    # secondary variables groups
    secgs = {'sig':('sx','sy','sz','sxy'),
             'eps':('ex','ey','ez','exy'),
             'sinvs':('p','q'), 'einvs':('ev','ed')}
    return ndim, nsov, vbcs, ebcs, secvs, secgs
```

and its constructor requires:

```python
class EelasticTri:
    def __init__(self, verts, params):
"""
Example of input:
          global_id tag    x    y
  verts = [[3,     -100, 0.0, 0.0],
           [4,     -100, 1.0, 0.0],
           [1,     -100, 0.0, 1.0]]
  params = {'E':1., 'nu':1.,
            'pstress':True,
            'thick':1., 'rho':1.}
  pstress : plane-stress instead
            of plane-strain?
            [optional]
  thick   : thickness for
            plane-stress
            only [optional]
```

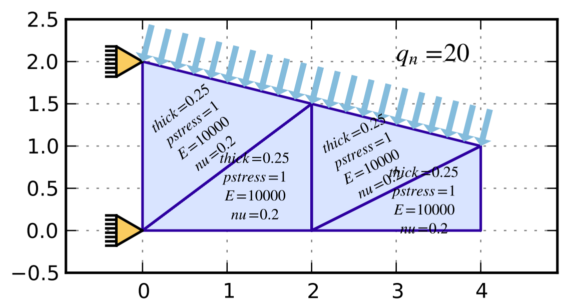

2D Elasticity ⇒ `EelasticTri`: example 1

Units
FEM Assembly
Plane Trusses
Plane Frames
2D Diffusion Tri
Stress/Strain
2D Elastic Tri
EelasticTri 1
EelasticTri 2
EelasticTri 3
EelasticTri 4
EelasticTri 5
EelasticTri 6
EelasticTri 7
EelasticTri 8
▷ Example 1 a
Example 1 b
Example 2 a
Example 2 b
Example 2 c
Example 2 d
Example 2 e
Example 2 f
Example 2 g
Example 2 h
Summary

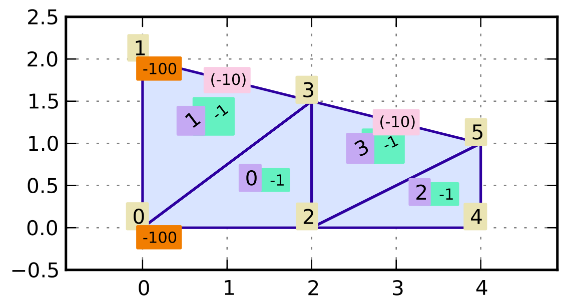
Example from [Bhatti 2005; p32; Example 1.6]

For the bracket shown to the right, find the nodal displacements and stresses at equilibrium state. The steps required are:

```python
from FEMsolver import *
# input mesh
V = [[0, -100, 0.0, 0.0],
     [1, -100, 0.0, 2.0],
     [2,    0, 2.0, 0.0],
     [3,    0, 2.0, 1.5],
     [4,    0, 4.0, 0.0],
     [5,    0, 4.0, 1.0]]
C = [[0, -1, [0,2,3]],
     [1, -1, [0,3,1], {1:-10}],
     [2, -1, [2,4,5]],
     [3, -1, [2,5,3], {1:-10}]]
m = FEMmesh(V, C)
m.draw();  m.show()
# parameters
p = {-1:{'E':1.0e+4, 'nu':0.2,
     'pstress':True, 'thick':0.25}}
```



The generated mesh:



Units
FEM Assembly
Plane Trusses
Plane Frames
2D Diffusion Tri
Stress/Strain
2D Elastic Tri
EelasticTri 1
EelasticTri 2
EelasticTri 3
EelasticTri 4
EelasticTri 5
EelasticTri 6
EelasticTri 7
EelasticTri 8
Example 1 a
▷ Example 1 b
Example 2 a
Example 2 b
Example 2 c
Example 2 d
Example 2 e
Example 2 f
Example 2 g
Example 2 h
Summary

```
# allocate fem solver object
s = FEMsolver(m, 'EelasticTri', p)

# boundary conditions
vb = {-100:{'ux':0.0, 'uy':0.0}}
eb = {-10:{'qnqt':(-20.0, 0.0)}}
s.set_bcs(eb=eb, vb=vb)

# solve (True => with reactions)
s.solve_steady(True)

# print displacements
s.print_u()

# print reactions
s.print_r()

# print stresses at elems
s.print_e()
```

The results are shown to the right (some output is truncated)

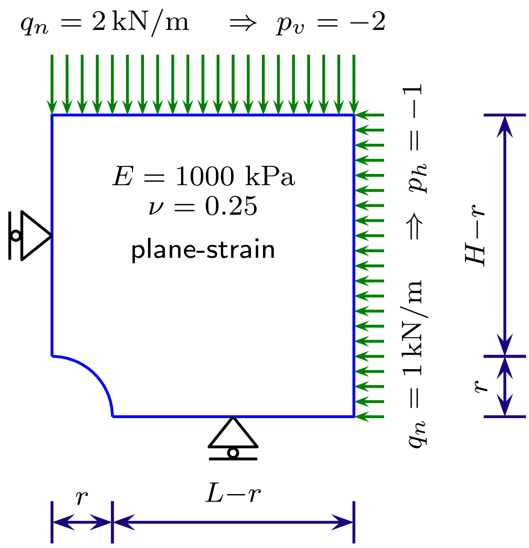
```
==============================
node          ux            uy
------------------------------
   0           0             0
   1           0             0
   2  -0.0103553    -0.0255297
   3  0.00472765    -0.0247357
   4  -0.0131394    -0.0554931
   5 8.38902e-05    -0.0555664
==============================
==============================
node         Rux           Ruy
------------------------------
   0       21.25       4.10648
   1      -16.25       15.8935
------------------------------
sum=           5            20
==============================
======================================================
elem         sx           sy            sz         sxy
------------------------------------------------------
   0   -52.8309     -5.27256             0    -11.2898
   1    24.6232      4.92464             0    -51.5326
   2   -14.6533     -3.66334             0    -7.32667
   3    3.10223      5.91407             0    -21.7822
======================================================
elem         ex           ey            ez         exy
------------------------------------------------------
   0 -0.00517764  0.000529362   0.00116207 -0.00135478
   1  0.00236383            0 -0.000590956 -0.00618391
   2 -0.00139207 -7.32667e-05  0.000366334  -0.0008792
   3 0.000191941  0.000529362 -0.000180326 -0.00261387
======================================================
```


Units
FEM Assembly
Plane Trusses
Plane Frames
2D Diffusion Tri
Stress/Strain
2D Elastic Tri
EelasticTri 1
EelasticTri 2
EelasticTri 3
EelasticTri 4
EelasticTri 5
EelasticTri 6
EelasticTri 7
EelasticTri 8
Example 1 a
Example 1 b
▷ Example 2 a
Example 2 b
Example 2 c
Example 2 d
Example 2 e
Example 2 f
Example 2 g
Example 2 h
Summary

The stress distribution around a deep underground opening, after excavation, can be approximated by Elasticity, as long as the material behaviour can be represented by a linear elastic model.

In this case, a simplified 2D plane-strain analysis can be carried out, where a $1\,\mathrm{m}$ thickness slice is considered and there is no displacements along the out-of-plane direction.

To illustrate, a quarter of a 2D section of a circle opening is shown to the right, taking advantage of symmetry.

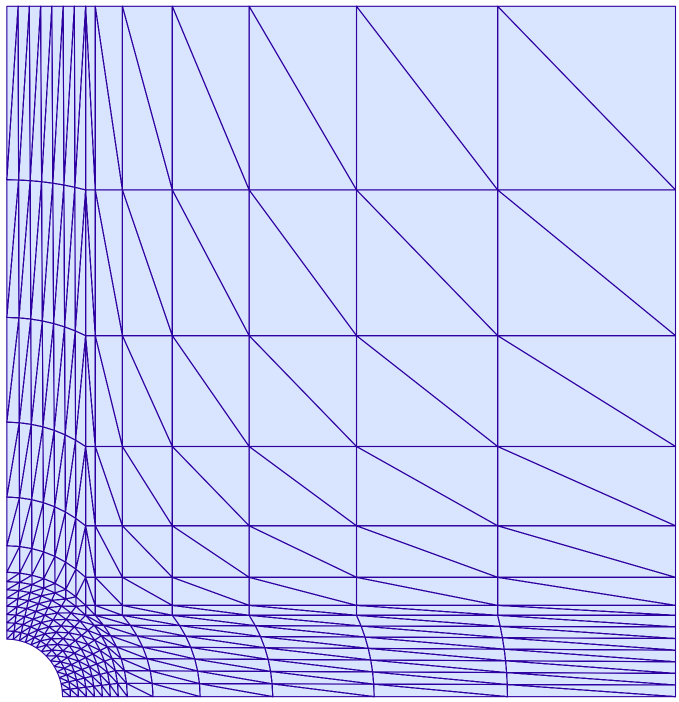


Find the stress distribution for $r = 0.5\,\mathrm{m}$ and $L = H = 6\,\mathrm{m}$

It is expected that the stresses concentrate near the circle opening while they approach the external loading at the extremities.

Therefore, a finite element mesh should be much finer around the opening. This strategy should always be employed in order to better capture higher gradients – remember that the strains are function of the gradient of displacements and stresses are function of strains.

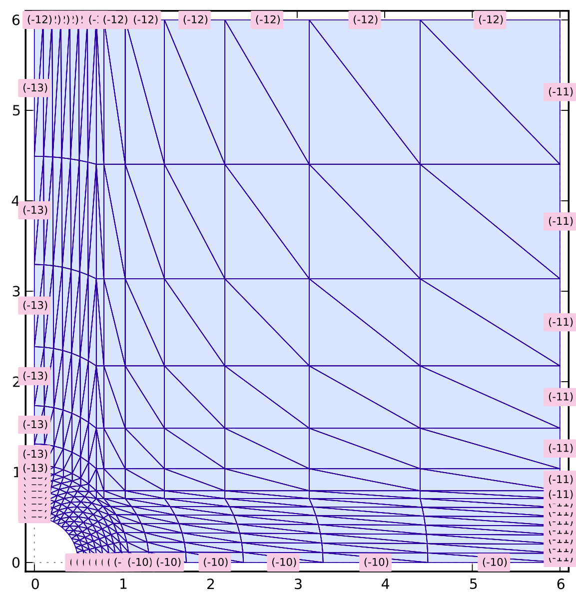
Mesh generated using an external software



Using `FEMsolver`, the numerical solution to this problem is easily obtained with:

```
# import solver
from FEMsolver import *
# import mesh
from qplateholeM1 import *
m = FEMmesh(V,C)
# input data
r,  L  = 0.5, 6.
ph, pv = -1., -2. ########### or: pv = -2.
# parameters and properties
p  = {-1:{'E':1000., 'nu':0.25}}
# solver
s  = FEMsolver(m, 'EelasticTri', p)
# boundary conditions
eb = {-10:{'uy':0.}, -11:{'qnqt':(ph,0.)},
      -13:{'ux':0.}, -12:{'qnqt':(pv,0.)}}
s.set_bcs(eb=eb)
# solve, extrapolate and generate results
s.solve_steady()
s.extrapolate()
s.write_vtu('qplatehole1.vtu')
```

The mesh with tagged edges is shown below:



Units
FEM Assembly
Plane Trusses
Plane Frames
2D Diffusion Tri
Stress/Strain
2D Elastic Tri
EelasticTri 1
EelasticTri 2
EelasticTri 3
EelasticTri 4
EelasticTri 5
EelasticTri 6
EelasticTri 7
EelasticTri 8
Example 1 a
Example 1 b
Example 2 a
Example 2 b
Example 2 c
▷ Example 2 d
Example 2 e
Example 2 f
Example 2 g
Example 2 h
Summary

The analytical solution to this problem is available (Kirsch's solution). In polar coordinates:

$$\sigma_r(d,\theta) = p_m\,(1-b) + p_d\,(1-4\,b+3\,b^2)\,\cos(2\,\theta)$$
$$\sigma_t(d,\theta) = p_m\,(1+b) - p_d\,(1+3\,b^2)\,\cos(2\,\theta)$$
$$\sigma_{rt}(d,\theta) = -p_d\,(1+2\,b-3\,b^2)\,\sin(2\,\theta)$$

where

$$p_m = \frac{ph+pv}{2} \qquad p_d = \frac{ph-pv}{2} \qquad b = \frac{r^2}{d^2}$$

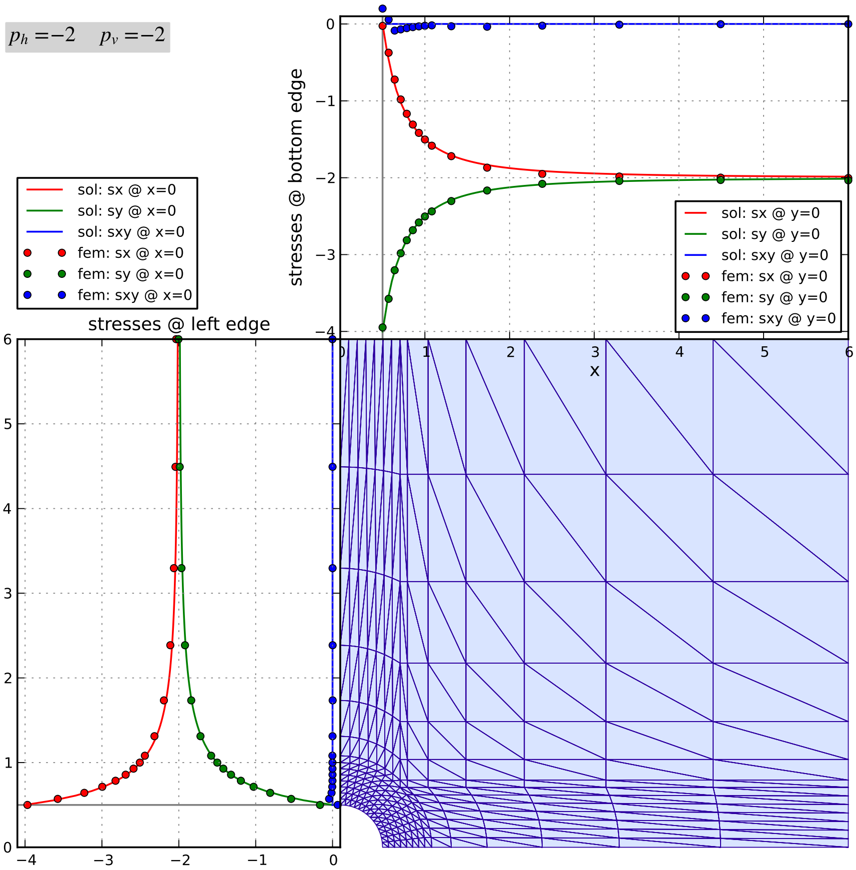
and

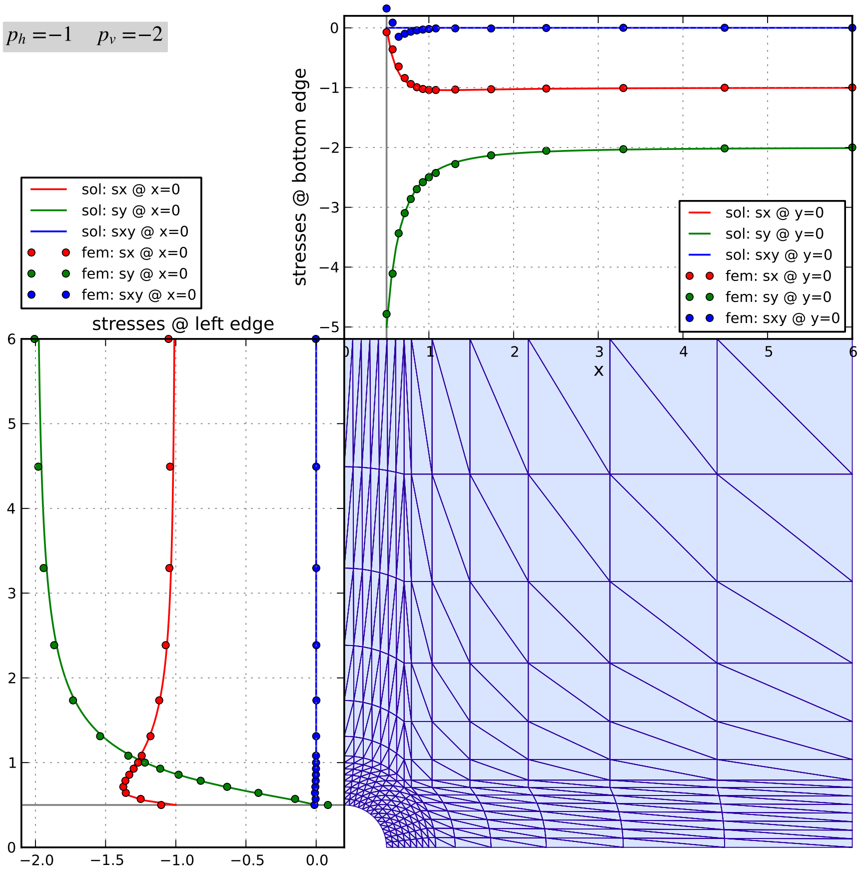
$$d = \sqrt{x^2+y^2} \qquad c = x/d \qquad s = y/d$$
$$\cos(2\,\theta) = c^2 - s^2 \qquad \sin(2\,\theta) = 2\,c\,s$$

to obtain the x-y stress components:

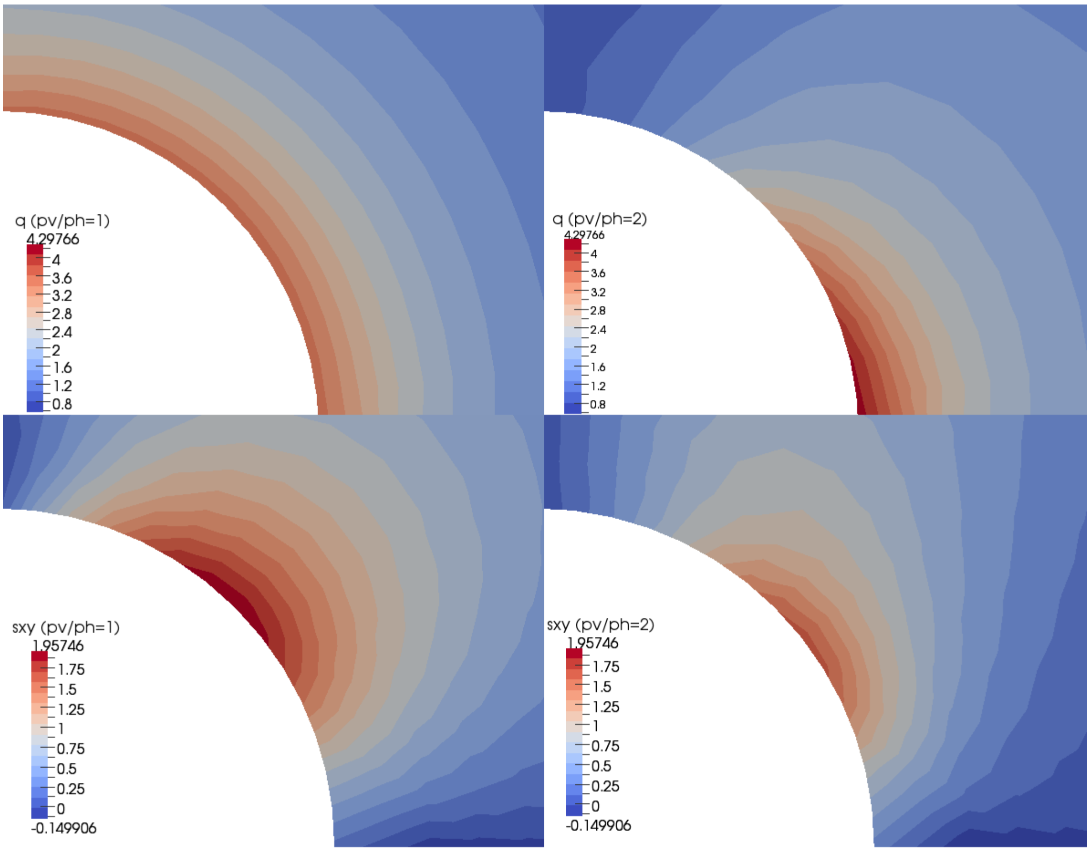
$$\sigma_x = c^2\,\sigma_r + s^2\,\sigma_t - 2\,c\,s\,\sigma_{rt}$$
$$\sigma_y = s^2\,\sigma_r + c^2\,\sigma_t + 2\,c\,s\,\sigma_{rt}$$
$$\sigma_{xy} = c\,s\,\sigma_r - c\,s\,\sigma_t + (c^2-s^2)\,\sigma_{rt}$$

Units
FEM Assembly
Plane Trusses
Plane Frames
2D Diffusion Tri
Stress/Strain
2D Elastic Tri
EelasticTri 1
EelasticTri 2
EelasticTri 3
EelasticTri 4
EelasticTri 5
EelasticTri 6
EelasticTri 7
EelasticTri 8
Example 1 a
Example 1 b
Example 2 a
Example 2 b
Example 2 c
Example 2 d
▷ Example 2 e
Example 2 f
Example 2 g
Example 2 h
Summary





$p_h = -1$ $p_v = -2$
stresses @ bottom edge
stresses @ left edge
x
sol: sx @ x=0
sol: sy @ x=0
sol: sxy @ x=0
fem: sx @ x=0
fem: sy @ x=0
fem: sxy @ x=0
sol: sx @ y=0
sol: sy @ y=0
sol: sxy @ y=0
fem: sx @ y=0
fem: sy @ y=0
fem: sxy @ y=0



q (pv/ph=1)
4.29766
q (pv/ph=2)
4.29766
sxy (pv/ph=1)
1.95746
-0.149906
sxy (pv/ph=2)
1.95746
-0.149906

2D Elasticity ⇒ EelasticTri: example 2

To generate the previous plots, the following commands can be used:

```
from msys_fig import *
# vertices ids and coordinates from FEM mesh
vh = m.get_verts_on_edges(-10, xsorted=True) # bottom verts
vv = m.get_verts_on_edges(-13, ysorted=True) # left vertices
X  = [m.V[i][2] for i in vh] # x coordinates of bottom verts
Y  = [m.V[i][3] for i in vv] # y coordinates of left vertices

# analytical solution
d = linspace(r, L, 101)
hstress = KirschStress(r,ph,pv, d, 0.) # y=0
vstress = KirschStress(r,ph,pv, 0., d) # x=0

# draw mesh
m.draw(ids=0,tags=0)              # draw without ids or tags
aa = gca()                        # grab current figure
axis('equal')                     # set equal units
aa.get_xaxis().set_visible(0)     # hide x scale
aa.get_yaxis().set_visible(0)     # hide y scale
aa.set_frame_on(0)                # hide frame

# divide figure into 3 parts
dv = make_axes_locatable(aa) # create a divider

# get handle to axes to plot bottom vertices stresses
ab = dv.append_axes("top", size=2.5,pad=0.,sharex=aa)

# get handle to axes to plot left vertices stresses
al = dv.append_axes("left",size=2.5,pad=0.,sharey=aa)
```

```
# y=0: bottom edge
sca(ab)                     # set current axes: ab
axvline(r,color='gray') # draw light vertical line
plot(d, hstress[0],color='r',label='sol: sx @ y=0')
plot(d, hstress[1],color='g',label='sol: sy @ y=0')
plot(d, hstress[2],color='b',label='sol: sxy @ y=0')
grid(color='gray')          # activate grid
plot(X, s.vvals['sx' ][vh], 'ro', label='fem: sx @ y=0')
plot(X, s.vvals['sy' ][vh], 'go', label='fem: sy @ y=0')
plot(X, s.vvals['sxy'][vh], 'bo', label='fem: sxy @ y=0')
axis([0.,L,-5.2,0.2])       # adjust limits
Gll('x','stresses @ bottom edge') # labels and legend

# x=0: left edge
sca(al)                     # set current axes: al
axhline(r,color='gray') # draw light horizontal line
plot(vstress[0], d, color='r', label='sol: sx @ x=0')
plot(vstress[1], d, color='g', label='sol: sy @ x=0')
plot(vstress[2], d, color='b', label='sol: sxy @ x=0')
grid(color='gray')          # activate grid
plot(s.vvals['sx' ][vv], Y, 'ro', label='fem: sx @ x=0')
plot(s.vvals['sy' ][vv], Y, 'go', label='fem: sy @ x=0')
plot(s.vvals['sxy'][vv], Y, 'bo', label='fem: sxy @ x=0')
axis([-2.1,0.2,0.,L])       # adjust limits
legend (bbox_to_anchor=(0.,1.06,1.,.102), loc=3, ncol=1,
        borderaxespad=0., handlelength=3, prop={'size':8})
title('stresses @ left edge',fontsize=10) # a title

# rescale and show
sca(aa)             # set current axes: mesh drawing
axis([0.,L,0.,L]) # adjust limits
show()              # show figure
```


Summary of four 2D FEM solutions: EelasticRod, EelasticBeam, EdiffusionTri, EelasticTri

Summary of FEM solutions: boundary conditions

Units
FEM Assembly
Plane Trusses
Plane Frames
2D Diffusion Tri
Stress/Strain
2D Elastic Tri
Summary
▷ Summary Bcs
Mesh Generation
Post-processing
References

To summarise, all boundary conditions that can be specified in the four 2D problems discussed here are listed in Table 3.

Table 3: Boundary conditions for some 2D problems.

Problem	**Essential: `utype`** (@ nodes)	**Natural: `ftype`** *concentrated* (@ nodes)	*distributed* (along edges)
Plane trusses	`ux uy`	`fx fy`	
Plane frames	`ux uy wz`	`fx fy mz`	`(qnl,qnr,qnt)`
2D Elasticity	`ux uy`	`fx fy`	`(qn,qt)`
2D Diffusion	`u`	`q`	`q c`

- **Note:** The setting up of essential values may be done using edge tags, i.e., setting *at edges*. The code will then transfer these values to all nodes on the corresponding edge
- **Note:** The essential values can be a function of x-y (space); ex: `{'u':lambda x,y:x+y}`. The natural values @ nodes can be a function of t (time); ex: `{'fy':lambda t:cos(t)}`

Mesh Generation

Units
FEM Assembly
Plane Trusses
Plane Frames
2D Diffusion Tri
Stress/Strain
2D Elastic Tri
Summary
Summary Bcs
▷ Mesh Generation
Post-processing
References

Global functions (not members of FEMmesh): Generate nonlinearly spaced points:

- `nlspace`

Mesh generation methods:

- `Gen1Dmesh`
- `Gen1Dlayered`
- `Gen2Dregion`
- `GenQplateHole`
- `GenQdisk`

Members of FEMmesh:

- `check_overlap`
- `tag_verts`
- `tag_vert`
- `tag_verts_on_line`
- `tag_edges_on_line`
- `get_verts`
- `get_verts_on_edges`

```
from FEMmesh  import * # import FEM mesh class
from msys_fig import * # import drawing routines
l1 = nlspace (0.0, 1.5, 5, n=0.5) # nonlinear spc
l2 = linspace(1.5, 2.0, 7)        # equally spc
l3 = (3.5-l1)[::-1]               # reversed pts
x  = hstack([l1,l2[1:],l3[1:]])   # x points
y  = nlspace (0.0, 2.0, 7, n=0.5) # y points
m  = Gen2Dregion(x,y)            # generate mesh
m.tag_edges_on_line(-22,1.75,0.,90.) # tag edges
m.tag_edges_on_line(-33,0.,1.155,0.,tol=0.001)
m.check_overlap()              # check overlaping
m.draw(ids=False)              #   nodes
Arrow(1.5, 2.3,1.5, 2.,zorder=15) # draw arrow
Arrow(2.0, 2.3,2.0, 2.,zorder=15) # draw arrow
Arrow(1.75,2.3,1.75,2.,zorder=15,fc='red')
m.show(); print x # show figure and print x vals
```

output:

```
[ 0.          0.75        1.06066017  1.29903811
  1.5         1.58333333  1.66666667  1.75
  1.83333333  1.91666667  2.          2.20096189
  2.43933983  2.75        3.5       ]
```

Post-processing: members of `FEMsolver`

Setting methods:

- `__init__`
- `set_bcs`

Solution methods:

- `solve_steady`

Post-processing methods:

- `get_u`
- `extrapolate`
- `write_vtu`
- `mesh_to_grid`

Printing and drawing methods:

- `print_u`
- `print_r`
- `print_e`
- `beam_plot`

Member variables with results:

- `U`
- `vvals`
- `React`

```
from FEMsolver import *          # import FEM solver
from msys_fig  import *          # msys drawing funcs
d = linspace(0.,1.,5)            # gen linear points
m = Gen2Dregion(d,d)             # generate 2D square
m.draw(0,0); m.show()            # (no ids or tags)
p = {-1:{'E':100.,'nu':0.25}} # parameters
s = FEMsolver(m,'EelasticTri',p) # solver
s.set_bcs(eb={  # set boundary conditions
  -10:{'ux':0.,'uy':0.},           # fix ux and uy
  -13:{'ux':0.,'uy':lambda x,y:0.},# fix with fcn
  -12:{'qnqt':(-1.,0.)}})          # distr. load
s.solve_steady() # solve equilibrium problem
s.extrapolate()  # => stresses available in vvals
s.write_vtu('elastic_example1') # for paraview
uvals = s.get_u()   # get u values by keys
print uvals['ux']   # print 'ux' values
print s.vvals['sx'] # print 'sx' values
s.mesh_to_grid() # convert nodal vals to grid vals
                 #   works only with rect. domains
Contour(s.xgrd,s.ygrd,s.gvals['sy']) # contour
show()                               # show figure
```


References

- **Bhatti**, M. A. (2005) Fundamental Finite Element Analysis and Applications: with Mathematica and Matlab Computations, John Wiley & Sons, Inc., Hoboken, New Jersey, 700p
- **Reddy**, J. N. (2006) An Introduction to the Finite Element Method, 3rd edition, McGraw-Hill, New York, 766p
- **Smith**, I. M. and **Griffiths**, D. V. (2006) Programming the Finite Element Method, 4th edition, John Wiley & Sons Ltd, Chinchester, 628p

Numerical Methods in Engineering – Part V

Dr Dorival Pedroso

May 24, 2012

Mesh Generation

Generating V and C with Blender

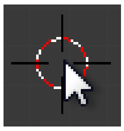
Mesh Generation
▷ Blender mesh 1
Blender mesh 2
Blender mesh 3
Blender mesh 4
Blender mesh 5
Blender mesh 6
Blender mesh 7
Blender mesh 8
Modelling
Post-processing
Eigenvalues
Time: 1st order
Time: 2nd order
Examples 1st o.
Examples 2nd o.

Blender (www.blender.org) is a very nice and powerful software for *3D modelling* and it can be extended by means of "addons" programmed in Python. To create meshes for `FEMsolver`, the addon `fem_blender_addon.py` can be used (from Blackboard). Its installation is illustrated in `blender_settingup.zip`

Generating V and C with Blender

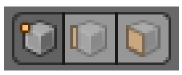
Mesh Generation
Blender mesh 1
▷ Blender mesh 2
Blender mesh 3
Blender mesh 4
Blender mesh 5
Blender mesh 6
Blender mesh 7
Blender mesh 8
Modelling
Post-processing
Eigenvalues
Time: 1st order
Time: 2nd order
Examples 1st o.
Examples 2nd o.

Mouse:



- `left-click` (or `lmb`) to re-position **input cursor** (very different from most CAD programs!)
- `middle-button-wheel-scroll` to zoom in/out
- `middle-button` to rotate 3D view (use `7` from numpad to get back to top view; press `5` to get an *ortho* view instead of perspective)
- `shift+middle-button` to pan

Object/edit mode:



- Use `tab` to swap between object and edit modes. Object mode places objects around (press `n` to see options). Edit mode allows mesh/geometry editing of: *Vertex*, *Edge*, *Face*. Activate selection with `ctrl+1` (2 or 3)

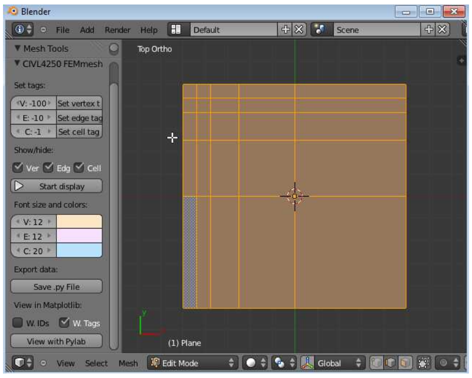
Example: (Sequence of keystrokes. Numbers in parenthesis are **png** files within `blender_example1.zip`)

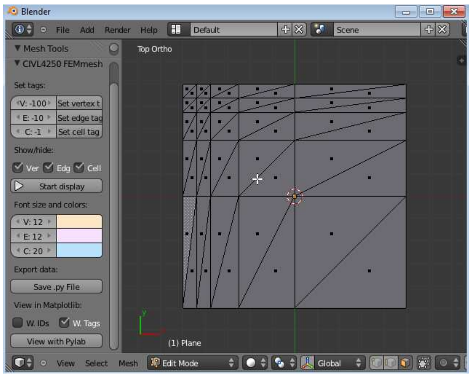
- ☐ `tab` to change to *edit mode* (**01, 02**)
- ☐ `shift+c` if you'd like to move the input cursor back to centre (**03**)
- ☐ `home a` to zoom and then deselect all
- ☐ `ctrl+tab+2` to activate edge-select
- ☐ right-click left edge to select (**04**)
- ☐ `g x 0.5 enter` to displace left edge 0.5 to the right (with x-direction fixed) (**05**)
- ☐ do the same: right-click right edge then: `g x -0.5 enter`
- ☐ do the same: right-click bottom edge then: `g y 0.5 enter`
- ☐ do the same: right-click top edge then: `g y -0.5 enter`
- ☐ `home` to zoom

- ☐ `ctrl+tab+3` to activate face-select
- ☐ `a` to select all (**06**)
- ☐ `w 1` to subdivide all edges (**07, 08**)
- ☐ `a` to deselect all
- ☐ `ctrl+tab+2` or edge-select
- ☐ Select left-most horizontal segments (can use `ctrl+left-click` for *lasso-select*) (**09, 10**)
- ☐ `w 1` to subdivide left horizontal edges (**11**)
- ☐ `a` to deselect all
- ☐ *lasso-select* left-most horizontal segments again (**12, 13**)
- ☐ `w 1` to subdivide (**14**)
- ☐ do the same with top-most vertical edges (**15-21**)

Generating V and C with Blender

- ☐ let's do 2 more subdivisions: one vertical and one horizontal (top- and left-most edges) (**22-26**)
- ☐ `a` to deselect all
- ☐ `ctrl+tab+3` to activate face-select
- ☐ `a` to select all faces (**27**)
- ☐ `ctrl+t` to convert quads to triangles (**28, 29**)
- ☐ `a` to select all (**30**)
- ☐ `w 4` to remove any doubles (overlapping vertices)
- ☐ `ctrl+s` to save (**31, 32**)
- ☐ **mesh is now ready to be "tagged"**





Generating V and C with Blender

Tagging:

- ☐ `ctrl+tab+2` to activate edge-select
- ☐ select bottom edges using `b` (box-select) (**33, 34**)
- ☐ click on *Set edge tag* (menu CIVL4250 FEMmesh) (**35**)
- ☐ activate visualisation of tags: click on *Start display* (**36, 37**)
- ☐ `ctrl+s` to save file
- ☐ `a` to deselect all
- ☐ `b` box-select right most edges (**38, 39**)
- ☐ click on top of the `E: -10` value and change the edge tag to `-11` (press `enter` to finish changing) (**40**)
- ☐ click on *Set edge tag* (**41**)
- ☐ tag all top edges with `-12` and the left edges with `-13` (**42**) (remember to press `a` to deselect previously selected edges)

Generating `V` and `C` with Blender

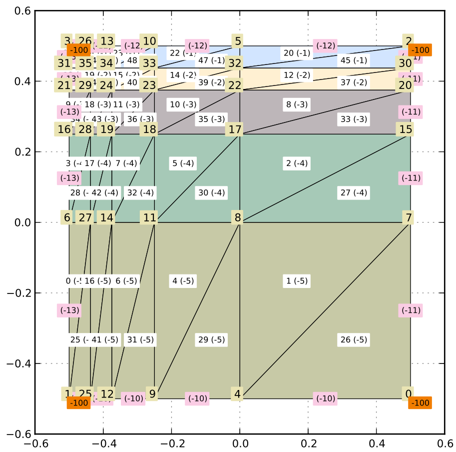
- ☐ `ctrl+s` to save file
- ☐ `a` to deselect all
- ☐ `ctrl+tab+1` to activate edge-select
- ☐ select the 4 corner vertices (with `right-click` and `shift`) (**43**)
- ☐ assign a `−100` tag to these vertices (**44**)
- ☐ `a` to deselect all
- ☐ `ctrl+tab+3` to activate face-select
- ☐ `a` to select all faces
- ☐ assign a `−1` tag to all faces (**45**)
- ☐ `ctrl+s` to save file
- ☐ click on *View with Pylab* in order to double check the mesh input. If it does not work, try to plot the mesh with a Python script in Eclipse (click on `Save .py File` to export V and C) (**46**)

Generating `V` and `C` with Blender

Notes:

- ☐ Use the code below in Eclipse to check your exported mesh
- ☐ To clear a tag, assign a zero-valued tag
- ☐ Any changes to the geometry **after** tags are assigned, may mess up a little with the tags. This happens because new vertices/edges may be created and then old ones loose their assignments. To fix this, re-assign the tags.
- ☐ It's good to use `w 4` to remove doubles (vertices) **before** tagging the mesh

```
from FEMmesh import *
from mesh1 import *
m = FEMmesh(V,C)
m.draw(); m.show()
```

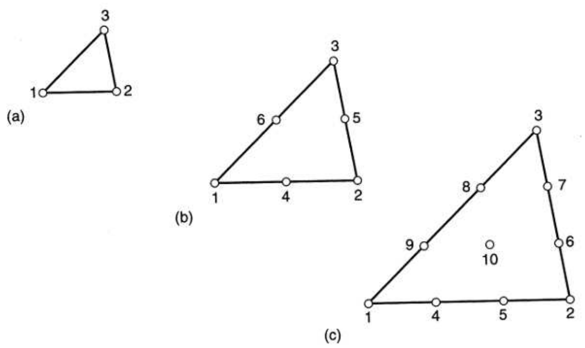


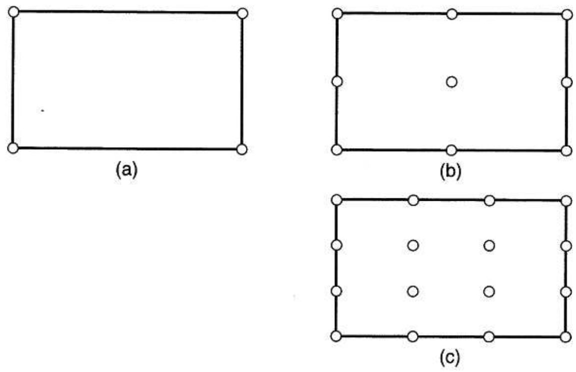
Note that in the mesh above, different tags were used for different layers of cells

Modelling Considerations

Mesh types

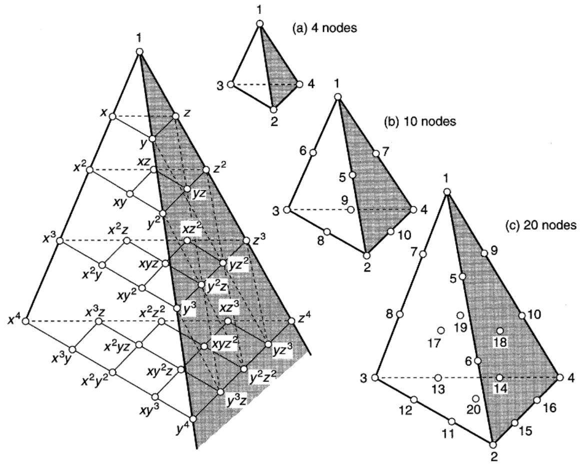
- ☐ Elements with different shapes can be developed
- ☐ Higher order elements are available ⇒ more nodes *per* element

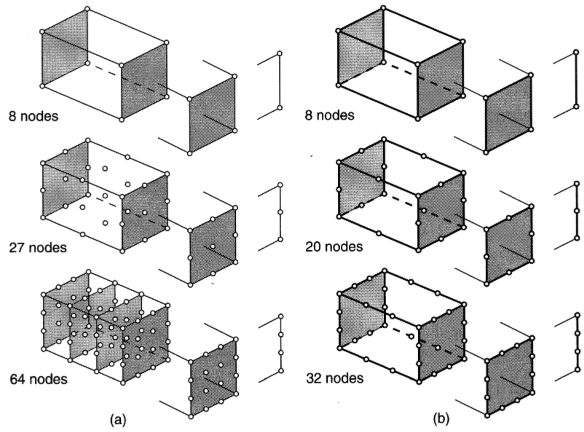




Mesh types

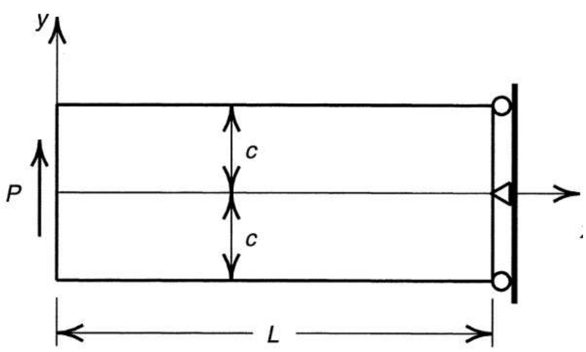
- Typical three-dimensional elements: tetrahedrons and hexahedrons

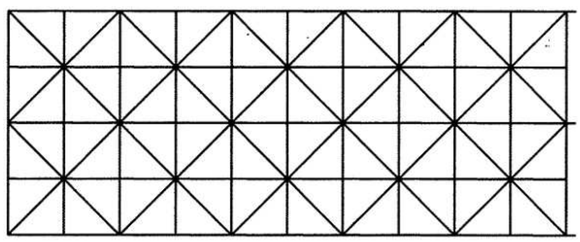




Convergence

- Convergence assessment: end loaded beam problem (Timoshenko & Goodier, 1970)

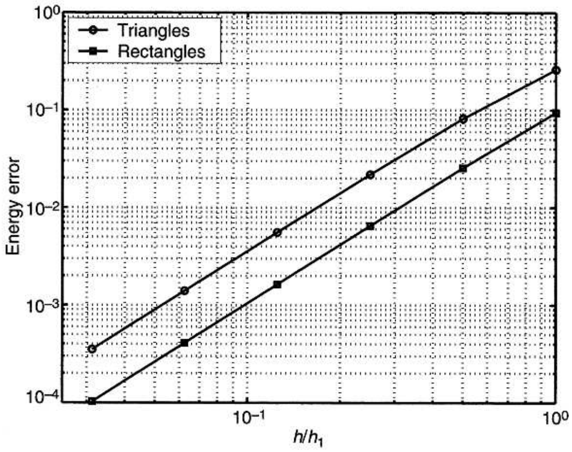


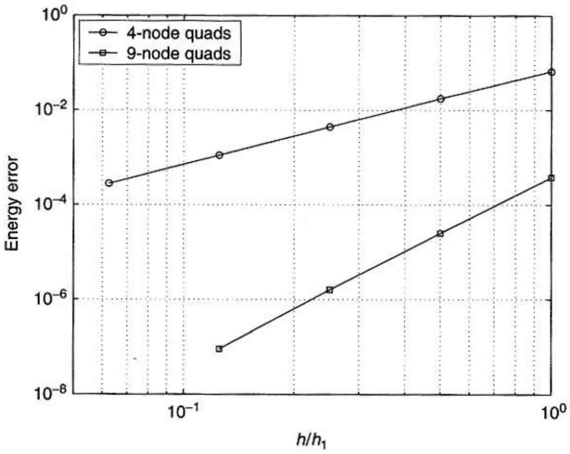


- End loaded beam results

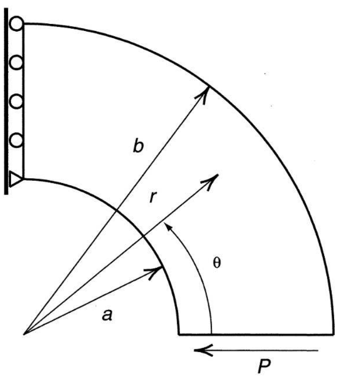
$$Energy = \boldsymbol{U}^T \boldsymbol{K} \boldsymbol{U}$$

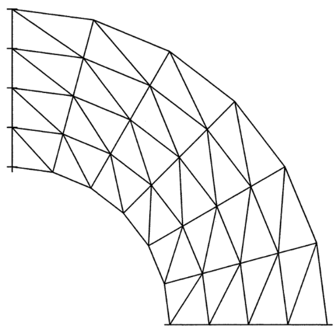
- h is a typical element size and h_1 is the typical element size of the coarse mesh

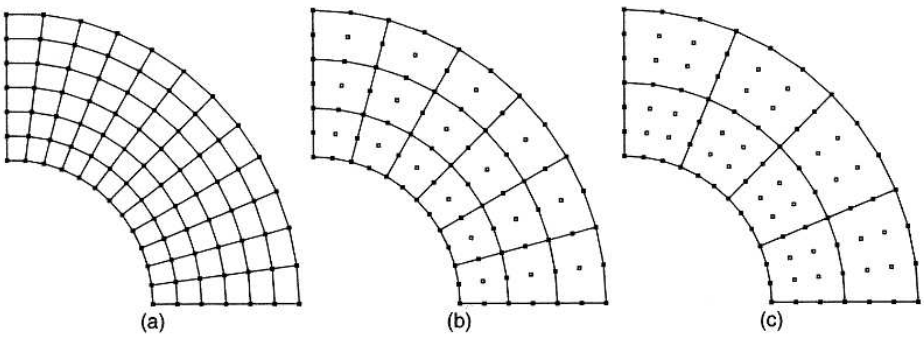




- Convergence assessment: end loaded circular beam





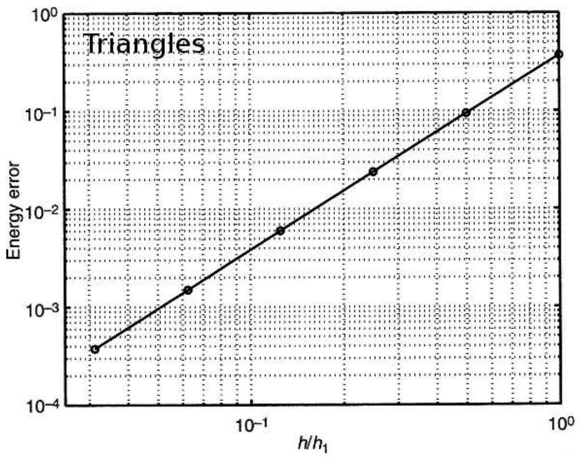


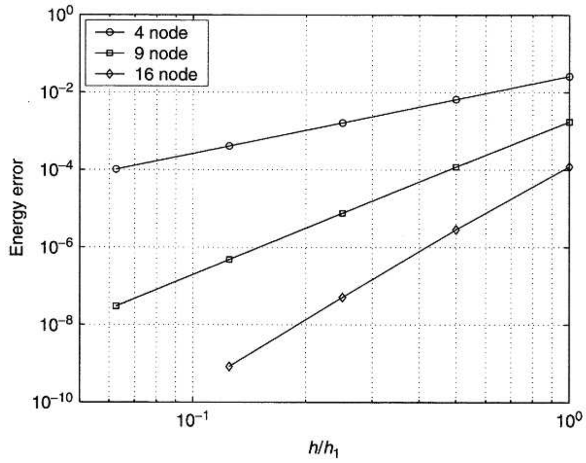
Mesh types

- ☐ End loaded circular beam results

$$Energy = \boldsymbol{U}^T \boldsymbol{K} \boldsymbol{U}$$

- ☐ h is a typical element size and h_1 is the typical element size of the coarse mesh





Summary on meshes and modelling considerations

Meshes:

- ☐ Elements must be compatible
- ☐ Elements must have "good" shapes (not elongated)
- ☐ Meshes should be finer closer to corners (higher gradients)

Simulations:

- ☐ Numerical simulations are approximations of the exact solution
- ☐ Engineering judgement is fundamental when modelling a problem (symmetries, boundary conditions, local refinements, etc.)
- ☐ Simulations must be verified by comparing with closed form solutions (analytical); for simpler problems, when possible
- ☐ Both numerical and analytical solutions must be validated against real observations, either by comparing with field data or laboratory experiments

Post-Processing

Members of `FEMsolver`

Setting methods:
- ☐ `__init__`
- ☐ `set_bcs`

Solution methods:
- ☐ `solve_steady`
- ☐ `solve_eigen`
- ☐ `solve_transient`
- ☐ `solve_dynamics`

Post-processing methods:
- ☐ `get_u`
- ☐ `get_mode`
- ☐ `calc_secondary`
- ☐ `extrapolate`
- ☐ `write_vtu`
- ☐ `mesh_to_grid`

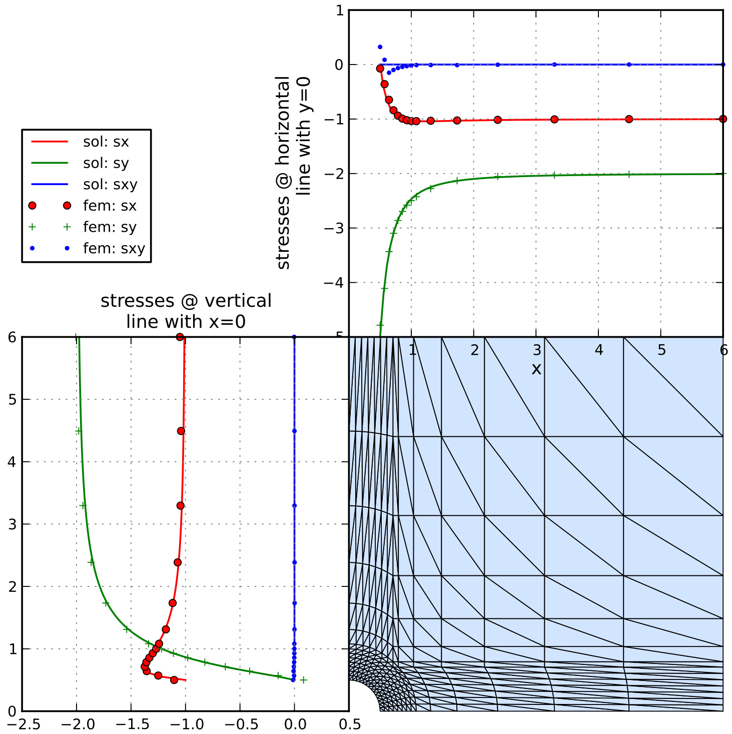
Internal methods:
- ☐ `out_results`

Printing and drawing methods:
- ☐ `print_u`
- ☐ `print_r`
- ☐ `print_e`
- ☐ `beam_plot`

Member variables with results:
- ☐ `U`
- ☐ `T`
- ☐ `UeqT`
- ☐ `Uout`
- ☐ `L` (after `eigen`)
- ☐ `Lv` (after `eigen`)
- ☐ `esvs` (after `calc_secondary`)
- ☐ `vvals` (after `extrapolate`)
- ☐ `React` (after `solve_steady`)

Plotting with Pylab (MatPlotLib): Example 1

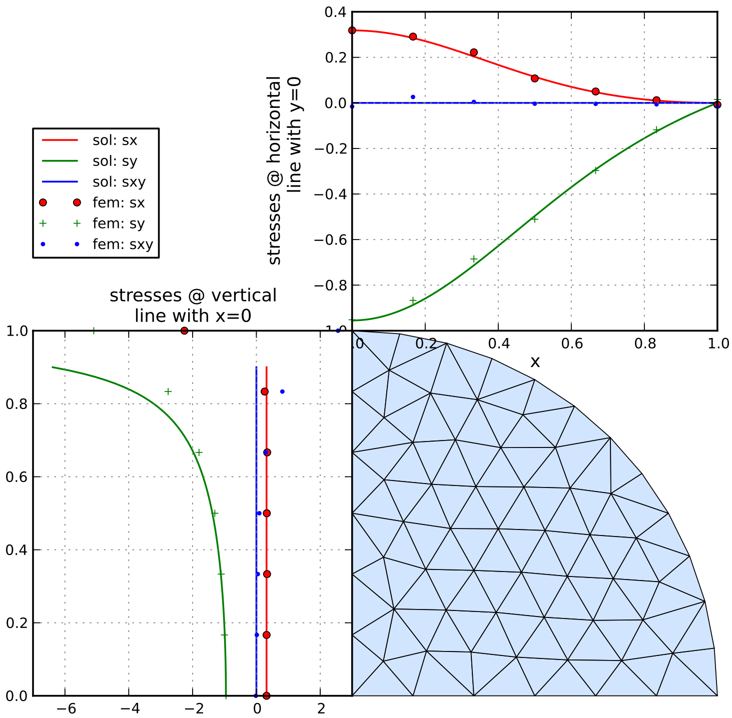
Quarter of a square region with a circle hole under distributed pressure to both sides (see file `qplatehole1.py` in Blackboard)



Plotting with Pylab (MatPlotLib): Example 2

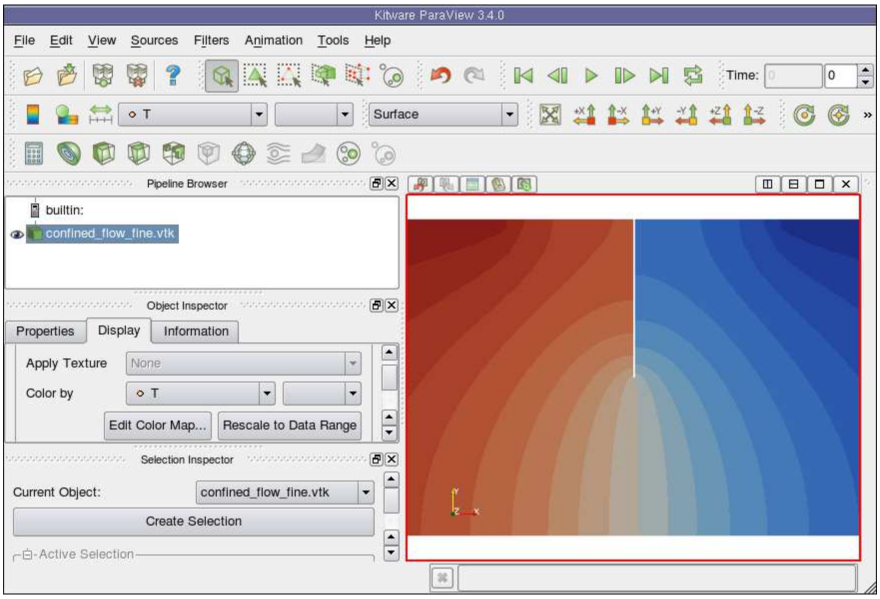
Mesh Generation
Modelling
Post-processing
FEMsolver
Pylab Plots 1
▷ Pylab Plots 2
ParaView 1
ParaView 2
ParaView 3
ParaView 4
ParaView 5
ParaView 6
Eigenvalues
Time: 1st order
Time: 2nd order
Examples 1st o.
Examples 2nd o.

Brazilian test: quarter of a disk with a concentrated force on its top-left corner (see file `braziliandisk1.py` in Blackboard)



Visualising with ParaView

ParaView (www.paraview.org) is a powerful tool for visualisation of results from finite element simulations. While plots can be easily created with Pylab, ParaView is more convenient when data attached to finite element meshes have to be processed. For example, the deformed mesh from elasticity solutions.



Visualising with ParaView

- ☐ The FEM can generate the following data: (a) scalar fields such as temperature, hydraulic head, vector fields; and (b) vector fields such as velocities and displacements
- ☐ In a FE mesh, data is available at nodes $\Rightarrow$ point data and eventually at the centre of elements $\Rightarrow$ cell data
- ☐ To convert from/to point/cell data, interpolation or extrapolation with (or not) averaging is needed

Typical methods of visualisation include:

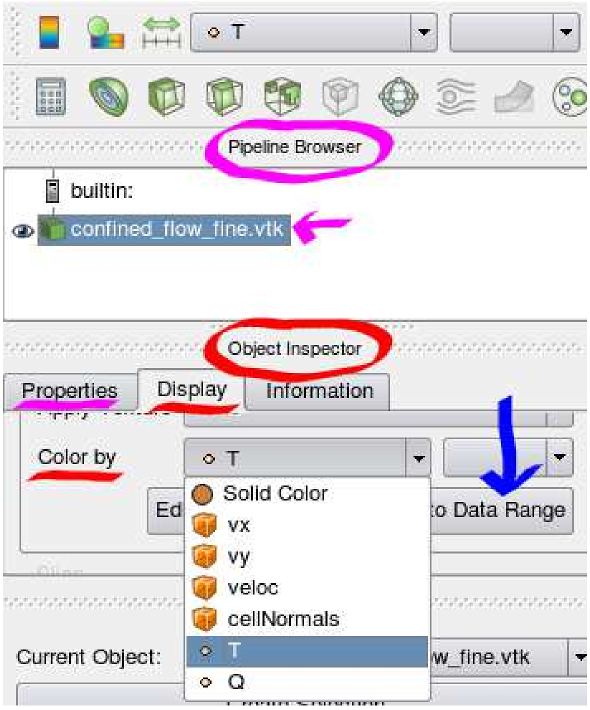
- ☐ Scalars: contours, cuts, x-y plots
- ☐ Vectors: glyphs, streamlines, ribbons

Visualisation often requires some data manipulation:

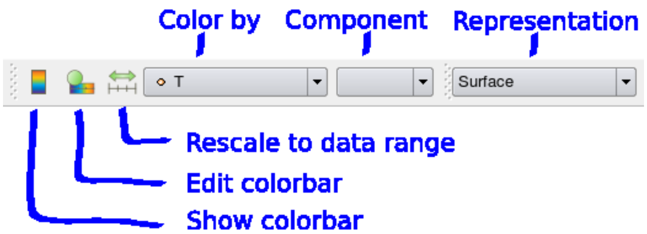
- ☐ Clip to a surface or volume
- ☐ Calculate new variables such as gradients, curvature, etc.

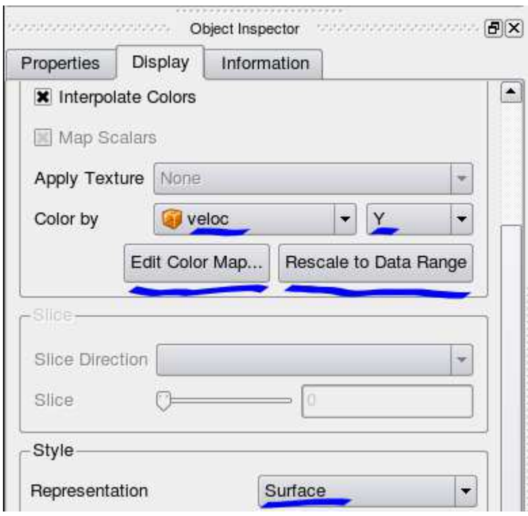
Example on how to draw *flowlines*:

- ☐ Load data file into ParaView: `File` ⇒ `Open` ⇒ `results.vtu`
- ☐ In the `Object Inspector` ⇒ `Properties`, click [Apply]
- ☐ In the `Pipeline Browser`, select `results.vtu` (data source)
- ☐ In the `Object Inspector` ⇒ `Display`, select `u` (total hydraulic head)
- ☐ Click on Rescale to Data Range



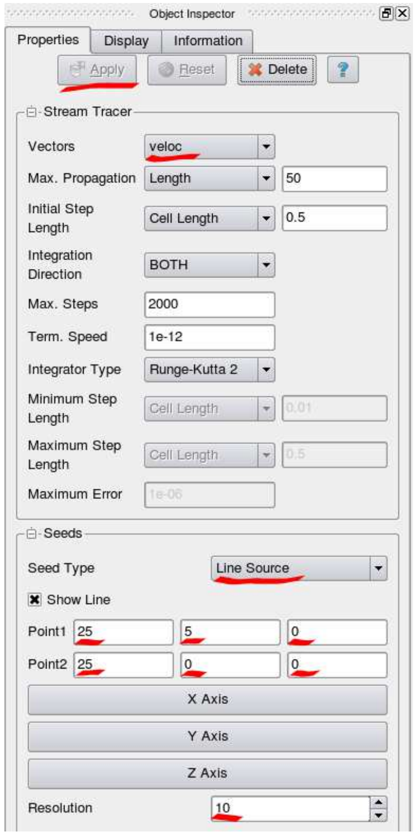
There is a convenient toolbar for accessing the most used options of the `Object Inspector`





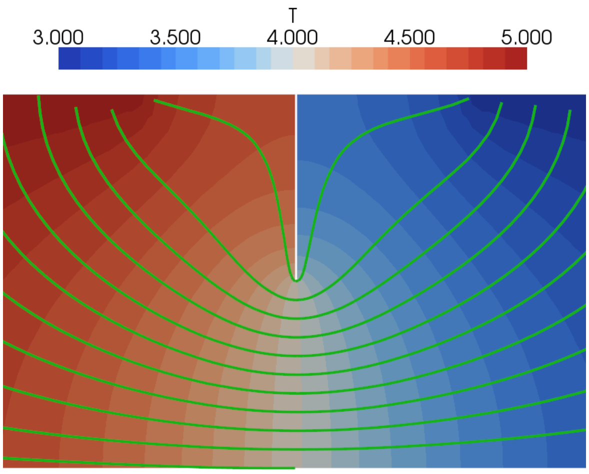
Visualising with ParaView

- ☐ Extrapolate "cell data" to "point data" (from the centre of elements to nodes): `Filters` ⇒ `Alphabetical` ⇒ `Cell Data to Point Data`
- ☐ In the `Object Inspector` ⇒ `Properties`, click [Apply]
- ☐ Create the streamlines: `Filters` ⇒ `Common` ⇒ `Stream Tracer`
- ☐ In the `Object Inspector` ⇒ `Properties`, set: Vectors = veloc; Seed Type = Line Source; Point1 = 25,5,0; Point2 = 25,0,0; and Resolution = 10
- ☐ Click [Apply]



Visualising with ParaView

Example of equipotential and flow lines:



Eigenvalue analyses

First order problems

After applying the FEM to the diffusion equation, the following element matrices are obtained:

$$\boldsymbol{C}^e\dot{\boldsymbol{U}}^e + \boldsymbol{K}^e\boldsymbol{U}^e = \boldsymbol{F}^e$$

The assemblage of all element equations leads to a *global system* of equations:

$$\boldsymbol{C}\dot{\boldsymbol{U}} + \boldsymbol{K}\boldsymbol{U} = \boldsymbol{F}$$

Further insights on this problem can be obtained by assuming the following k number of possible solutions $\boldsymbol{U}_k$:

$$\boldsymbol{U}_k = e^{-\mu_k\, t}\,\boldsymbol{b} \quad \text{with} \quad \mu_k > 0$$

where $\boldsymbol{b}$ is a vector of some (to be determined) constant values, i.e. it's time independent. Hence:

$$\dot{\boldsymbol{U}}_k = -\mu_k\, e^{-\mu_k\, t}\,\boldsymbol{b} = -\mu_k \boldsymbol{U}_k \qquad \text{no sum on } k$$

First order problems

Therefore, the global system becomes:

$$-\mu_k \boldsymbol{C}\boldsymbol{U}_k + \boldsymbol{K}\boldsymbol{U}_k = \boldsymbol{0}$$

or

$$\boxed{\boldsymbol{K}\boldsymbol{U}_k = \mu_k \boldsymbol{C}\boldsymbol{U}_k}$$

resulting in a typical **generalised eigenvalue problem**, where μ_k are the eigenvalues and $\boldsymbol{U}_k$ are the eigenvectors.

Note that μ_k are positive and real, as long as $\boldsymbol{K}$ and $\boldsymbol{C}$ are positive-definite (the case here).

Note also that the problem of updating $\dot{\boldsymbol{U}}_k = -\mu_k \boldsymbol{U}_k$ in time is similar to our previous scalar problem: $\dot{u} = \lambda\, u$, if we set $\lambda_k = -\mu_k$.

Second order problems

The assemblage of all element equations in a dynamics problem leads to a *global system* of equations:

$$\boldsymbol{M}\ddot{\boldsymbol{U}} + \boldsymbol{C}\dot{\boldsymbol{U}} + \boldsymbol{K}\boldsymbol{U} = \boldsymbol{F}$$

For zero *damping* matrix $\boldsymbol{C}$ and homogeneous problem $\boldsymbol{F} = \boldsymbol{0}$:

$$\boldsymbol{M}\ddot{\boldsymbol{U}} + \boldsymbol{K}\boldsymbol{U} = \boldsymbol{0}$$

Assuming a number of possible solutions $\boldsymbol{U}_k$ given by:

$$\boldsymbol{U}_k = e^{i\,\omega_k\, t}\,\boldsymbol{b} \quad \text{with} \quad \omega_k > 0$$

where $i = \sqrt{-1}$ and $\boldsymbol{b}$ is a vector of some (to be determined) constant values (time independent).
Hence:

$$\dot{\boldsymbol{U}}_k = i\,\omega_k\, e^{i\,\omega_k\, t}\,\boldsymbol{b} = i\,\omega_k \boldsymbol{U}_k \qquad \text{no sum on } k$$

And: $$\ddot{\boldsymbol{U}}_k = i\,\omega_k\,(i\,\omega_k)\,\boldsymbol{U}_k = -\omega_k^2\boldsymbol{U}_k \qquad \text{no sum on } k$$

Therefore, the global system becomes:

$$-\omega_k^2\,\boldsymbol{M}\boldsymbol{U}_k + \boldsymbol{K}\boldsymbol{U}_k = \boldsymbol{0}$$

or

$$\boxed{\boldsymbol{K}\boldsymbol{U}_k = \lambda_k\,\boldsymbol{M}\boldsymbol{U}_k}$$

resulting in a typical **generalised eigenvalue problem**, where $\lambda_k = \omega_k^2$ are the eigenvalues and $\boldsymbol{U}_k$ are the eigenvectors.

Note that λ_k are positive and real, as long as $\boldsymbol{K}$ and $\boldsymbol{M}$ are positive-definite (the case here).

Time approximation methods: First Order Problems

As we have seen before, first order differential equations representing an initial value problem (IVP) can be approximated by finite differences. In particular, the Euler/forward can be made stable after finding the critical step size and the Euler/backward is unconditionally stable. On the other hand, the central difference is **never stable** for first order IVPs.

For convenience, the expressions to update u from u_i to u_{i+1} when the time is incremented from $t_i = t$ to $t_{i+1} = t + h$ are reviewed below:

$$\text{Euler/forward:} \qquad u_{i+1} = u_i + h\ \dot{u}|_{t_i}$$

$$\text{Euler/backward:} \qquad u_{i+1} = u_i + h\ \dot{u}|_{t_{i+1}}$$

where u is some scalar quantity and $\dot{u} = \frac{\mathrm{d}u}{\mathrm{d}t} = f(t, u)$.

Both the Euler forward and backward methods are first order accurate. It is desired then to develop more accurate methods. One first idea is to mix the forward and backward terms in the same expression. This leads to the so-called θ-method where θ works as a weight applied to each one forward/explicit or backward/implicit term.

The θ-method is simply given by:

$$u_{i+1} = u_i + (1-\theta)\, h\, \dot{u}_i + \theta\, h\, \dot{u}_{i+1}$$

where: $\dot{u}_i = \dot{u}|_{t_i}$ and $\dot{u}_{i+1} = \dot{u}|_{t_{i+1}}$

Note that the Euler/forward method is recovered with $\theta = 0$ and the backward method with $\theta = 1$.

To investigate the stability properties of the θ-method, let's consider Dahlquist's equation again:

$$\frac{\mathrm{d}u}{\mathrm{d}t} = \lambda\, u \quad \text{with} \quad u_0 = 1 \quad \text{and} \quad \lambda < 0$$

Remember that the solution to this IVP is $u(t) = e^{\lambda\, t}$ and $\lambda < 0$ because we require that the solution converges to a final value after large t. The θ method applied to the above equation is then:

$$u_{i+1} = u_i + (1-\theta)\, h\, \underbrace{\lambda\, u_i}_{\dot{u}_i} + \theta\, h\, \underbrace{\lambda\, u_{i+1}}_{\dot{u}_{i+1}}$$

θ-method: scalar first-order equation

Solving for u_{i+1}:

$$u_{i+1} = \frac{1 + (1-\theta)\,\lambda\,h}{1 - \theta\,\lambda\,h}\,u_i$$

After many (n) updates:

$$u_n = \left[\frac{1 + (1-\theta)\,\lambda\,h}{1 - \theta\,\lambda\,h}\right]^n u_0$$

Therefore, for stability, the following condition must be satisfied:

$$\left|\frac{1 + (1-\theta)\,\lambda\,h}{1 - \theta\,\lambda\,h}\right| \leq 1$$

$$-1 \leq \frac{1 + (1-\theta)\,\lambda\,h}{1 - \theta\,\lambda\,h} \leq 1$$

$$\theta\,\lambda\,h - 1 \leq 1 + (1-\theta)\,\lambda\,h \leq 1 - \theta\,\lambda\,h$$

$$\theta\,\lambda\,h - 2 \leq (1-\theta)\,\lambda\,h \leq -\theta\,\lambda\,h$$

θ-method: scalar first-order equation

Noting that $-\lambda = |\lambda|$, since $\lambda < 0$, then:

$$(1-\theta)\,\lambda\,h \geq \theta\,\lambda\,h - 2 \qquad \text{and} \qquad (1-\theta)\,\lambda\,h \leq \theta\,|\lambda|\,h$$

The second condition is always satisfied since $0 \leq \theta \leq 1$. The first one leads to:

$$(1-\theta)\,\lambda\,h \geq \theta\,\lambda\,h - 2 \qquad \times(-1)$$

$$(1-\theta)\,(-\lambda)\,h \leq \theta\,(-\lambda)\,h + 2$$

$$(1-\theta)\,|\lambda|\,h - \theta\,|\lambda|\,h \leq 2$$

thus:

$$\boxed{h \leq \frac{2}{(1-2\,\theta)|\lambda|}}$$

Since h has to be positive, the stability criterion applies only to methods with $\theta < 1/2$. Thus, all methods with $\theta \geq 1/2$ are *unconditionally stable*.

θ-method: scalar first-order equation

Mesh Generation
Modelling
Post-processing
Eigenvalues
Time: 1st order
θ-method 1
θ-method 2
θ-method 3
θ-method 4
▷ θ-method 5
Time: 2nd order
Examples 1st o.
Examples 2nd o.

It can be shown that the stability of the θ-method, when applied to $\boldsymbol{C}\dot{\boldsymbol{U}} + \boldsymbol{K}\boldsymbol{U} = \boldsymbol{0}$, requires that:

$$\boxed{h \leq \frac{2}{(1 - 2\,\theta)\max(\mu_k)}}$$

where h is the time step and $\max(\mu_k)$ indicates the maximum value of the eigenvalues μ_k. Of course, this is only required when $\theta < 1/2$.

Note that the stability analysis of the θ-method with $\theta < 1/2$ is developed for the *homogeneous* form of the governing equation (with $\boldsymbol{F} = \boldsymbol{0}$) and that the essential boundary conditions must be applied before the study.
$\theta = 1/4 \Rightarrow$ Crank-Nicolson's method

Mesh Generation
Modelling
Post-processing
Eigenvalues
Time: 1st order
▷ Time: 2nd order
Newmark 1
Newmark 2
Newmark 3
Newmark 4
Newmark 5
Newmark 6
Newmark 7
HHT 1
HHT 2
FEM equations 1
FEM equations 2
Rayleigh 1
Examples 1st o.
Examples 2nd o.

Time approximation methods: Second Order Problems

Newmark's method

Suppose we have to solve the following system of second order differential equations:

$$\boldsymbol{M}\,\ddot{\boldsymbol{u}} + \boldsymbol{C}\,\dot{\boldsymbol{u}} + \boldsymbol{K}\,\boldsymbol{u} = \boldsymbol{g}$$

or the following two systems of first order equations:

$$\begin{aligned} \dot{\boldsymbol{u}} &= \boldsymbol{v} \\ \boldsymbol{M}\,\dot{\boldsymbol{v}} + \boldsymbol{C}\,\boldsymbol{v} + \boldsymbol{K}\,\boldsymbol{u} &= \boldsymbol{g} \end{aligned}$$

As with the θ-method, the (generalised) Newmark's method employs some sort of weighted Taylor's expansion to update $\boldsymbol{v}_i = \boldsymbol{v}_0$ and $\boldsymbol{u}_i = \boldsymbol{u}_0$, from $t_i = t_0$ to $t_{i+1} = t_1 = t_0 + h$, in order to obtain $\boldsymbol{v}_{i+1} = \boldsymbol{v}_1$ and $\boldsymbol{u}_{i+1} = \boldsymbol{u}_1$. The method can then be written as:

$$\begin{aligned} \boldsymbol{v}_1 &= \boldsymbol{v}_0 + (1-\theta_1)\,h\,\dot{\boldsymbol{v}}_0 + \theta_1\,h\,\dot{\boldsymbol{v}}_1 \\ \boldsymbol{u}_1 &= \boldsymbol{u}_0 + h\,\boldsymbol{v}_0 + (1-\theta_2)\,H\,\dot{\boldsymbol{v}}_0 + \theta_2\,H\,\dot{\boldsymbol{v}}_1 \end{aligned}$$

where $H = h^2/2$. The standard Newmark parameters are $\gamma = \theta_1$ and $\beta = \theta_2/2$.

Newmark's method

Note that when $\theta_1 = 0$ and $\theta_2 = 0$ the fundamental kinematic expressions considering **constant acceleration** are recovered:

$$\begin{aligned} \boldsymbol{v}_1 &= \boldsymbol{v}_0 + \boldsymbol{a}_0\,\Delta t \\ \boldsymbol{u}_1 &= \boldsymbol{u}_0 + \boldsymbol{v}_0\,\Delta t + \boldsymbol{a}_0\,\tfrac{\Delta t^2}{2} \end{aligned}$$

where $\boldsymbol{a} = \dot{\boldsymbol{v}}$ and $h = \Delta t$.

Now, together with the governing expression evaluated at the *future* state:

$$\boldsymbol{M}\,\dot{\boldsymbol{v}}_1 + \boldsymbol{C}\,\boldsymbol{v}_1 + \boldsymbol{K}\,\boldsymbol{u}_1 = \boldsymbol{g}_1$$

the two expressions:

$$\begin{aligned} \boldsymbol{v}_1 &= \boldsymbol{v}_0 + (1-\theta_1)\,h\,\dot{\boldsymbol{v}}_0 + \theta_1\,h\,\dot{\boldsymbol{v}}_1 \\ \boldsymbol{u}_1 &= \boldsymbol{u}_0 + h\,\boldsymbol{v}_0 + (1-\theta_2)\,H\,\dot{\boldsymbol{v}}_0 + \theta_2\,H\,\dot{\boldsymbol{v}}_1 \end{aligned}$$

can be solved for $\boldsymbol{u}_1$, $\boldsymbol{v}_1$, and $\dot{\boldsymbol{v}}_1$ (3 expressions, 3 unknowns).

Newmark's method

Then, to minimize matrix-vector operations, the following procedure is taken: first we solve for $\dot{\boldsymbol{v}}_1$ (from the second expression – for $\boldsymbol{u}_1$):

$$\dot{\boldsymbol{v}}_1 = \underbrace{\frac{1}{\theta_2\, H}}_{\alpha_1} \boldsymbol{u}_1 - \underbrace{\frac{1}{\theta_2\, H}}_{\alpha_1} \boldsymbol{u}_0 - \underbrace{\frac{h}{\theta_2\, H}}_{\alpha_2} \boldsymbol{v}_0 - \underbrace{\frac{(1-\theta_2)}{\theta_2}}_{\alpha_3} \dot{\boldsymbol{v}}_0$$

or:

$$\boxed{\dot{\boldsymbol{v}}_1 = \alpha_1\, \boldsymbol{u}_1 - \boldsymbol{p}}$$

with:

$$\boxed{\boldsymbol{p} = \alpha_1\, \boldsymbol{u}_0 + \alpha_2\, \boldsymbol{v}_0 + \alpha_3\, \dot{\boldsymbol{v}}_0}$$

and (noting that $H = h^2/2$):

$$\boxed{\alpha_1 = \frac{1}{\theta_2\, H} \qquad \alpha_2 = \frac{h}{\theta_2\, H} \qquad \alpha_3 = \frac{1}{\theta_2} - 1}$$

Now, inserting this equation for $\dot{\boldsymbol{v}}_1$ into the first expression for $\boldsymbol{v}_1$:

$$\boldsymbol{v}_1 = \boldsymbol{v}_0 + (1-\theta_1)\, h\, \dot{\boldsymbol{v}}_0 + \frac{\theta_1\, h}{\theta_2\, H}\boldsymbol{u}_1 - \frac{\theta_1\, h}{\theta_2\, H}\boldsymbol{u}_0 - \frac{\theta_1\, h^2}{\theta_2\, H}\boldsymbol{v}_0 - \frac{\theta_1\, h\,(1-\theta_2)}{\theta_2}\dot{\boldsymbol{v}}_0$$

Newmark's method

Noting that $H = h^2/2$, and re-arranging:

$$\boldsymbol{v}_1 = \underbrace{\frac{\theta_1\, h}{\theta_2\, H}}_{\alpha_4} \boldsymbol{u}_1 - \underbrace{\frac{\theta_1\, h}{\theta_2\, H}}_{\alpha_4} \boldsymbol{u}_0 - \underbrace{\left(\frac{2\,\theta_1}{\theta_2} - 1\right)}_{\alpha_5} \boldsymbol{v}_0 - \underbrace{\left(\frac{\theta_1}{\theta_2} - 1\right) h}_{\alpha_6}\, \dot{\boldsymbol{v}}_0$$

or:

$$\boxed{\boldsymbol{v}_1 = \alpha_4\, \boldsymbol{u}_1 - \boldsymbol{q}}$$

with:

$$\boxed{\boldsymbol{q} = \alpha_4\, \boldsymbol{u}_0 + \alpha_5\, \boldsymbol{v}_0 + \alpha_6\, \dot{\boldsymbol{v}}_0}$$

and:

$$\boxed{\alpha_4 = \frac{\theta_1\, h}{\theta_2\, H} \qquad \alpha_5 = \frac{2\,\theta_1}{\theta_2} - 1 \qquad \alpha_6 = \left(\frac{\theta_1}{\theta_2} - 1\right) h}$$

Note that this works only for $\theta_2 \neq 0$. For $\theta_2 = 0$, another derivation has to be considered. Actually, it does not really matter which among $\boldsymbol{u}_1$, $\boldsymbol{v}_1$, and $\dot{\boldsymbol{v}}_1$ is solved first.

Newmark's method

Now, inserting $\dot{\boldsymbol{v}}_1$ and $\boldsymbol{u}_1$ into the governing equation:

$$\boldsymbol{M}\,(\alpha_1\,\boldsymbol{u}_1 - \boldsymbol{p}) + \boldsymbol{C}\,(\alpha_4\,\boldsymbol{u}_1 - \boldsymbol{q}) + \boldsymbol{K}\,\boldsymbol{u}_1 = \boldsymbol{g}_1$$

we can finally solve for $\boldsymbol{u}_1$:

$$\boxed{\boldsymbol{u}_1 = (\alpha_1\,\boldsymbol{M} + \alpha_4\,\boldsymbol{C} + \boldsymbol{K})^{-1}\,(\boldsymbol{g}_1 + \boldsymbol{M}\,\boldsymbol{p} + \boldsymbol{C}\,\boldsymbol{q})}$$

Newmark's method

It can be shown that the Generalised Newmark method is *unconditionally stable* for:

$$\boxed{\frac{1}{2} \leq \theta_1 \leq \theta_2}$$

Otherwise, for $\theta_1 \geq 1/2$ and $\theta_2 \leq \theta_1$ the time step required for stability is constrained by:

$$\boxed{\Delta t \leq \frac{\sqrt{2}}{\omega_{max}\sqrt{\theta_1 - \theta_2}} \quad \text{with} \quad \theta_1 \geq 1/2 \quad \text{and} \quad \theta_2 \leq \theta_1}$$

where ω_{max} is the maximum natural frequency of the corresponding system, including the boundary conditions and without the damping matrix.

Note however that errors due to *numerical damping* and/or *period elongation* may happen for $\theta_1 > 1/2$.

For very large systems, where the eigenvalue analysis may become too slow, another direct method to estimate the critical time step is desired. A method that considers an approximation of the smallest period for a given finite element discretisation T_{min} is given by:

$$\Delta t \leq \frac{T_{min}\sqrt{2}}{2\,\pi\sqrt{\theta_1 - \theta_2}} \quad \text{with} \quad \theta_1 \geq 1/2 \quad \text{and} \quad \theta_2 \leq \theta_1$$

Nonetheless, the evaluation of T_{min} is not an easy procedure and an alternative estimate has to be considered. By doing so, for linear elastic problems, the critical time step is approximated by:

$$\boxed{\Delta t \leq \frac{2\,h_{min}}{\sqrt{3}\,c} \quad \text{with} \quad \theta_1 \geq 1/2, \quad \theta_2 \leq \theta_1 \quad \text{and} \quad c = \sqrt{E/\rho}}$$

where c is the *speed of elastic wave propagation* and h_{min} is the smallest (characteristic) element size in the whole mesh. It can be seen that the critical time step decreases for finer meshes and for higher elastic wave speeds.

Sometimes, it is interesting to artificially introduce some damping during the numerical solution. This aims to to compensate for some excessive oscillation produced by the numerical scheme. To do so, the Newmark's method was slightly modified by Hilber, Hughes and Taylor resulting on the so-called HHT method (or α-method). The only difference now is the inclusion of some averaged terms for the velocity and displacements in the governing equation:

$$\boldsymbol{M}\,\dot{\boldsymbol{v}}_1 + (1+\alpha)\,\boldsymbol{C}\,\boldsymbol{v}_1 - \alpha\,\boldsymbol{C}\,\boldsymbol{v}_0 + (1+\alpha)\,\boldsymbol{K}\,\boldsymbol{u}_1 - \alpha\,\boldsymbol{K}\,\boldsymbol{u}_0 = \boldsymbol{g}_\alpha$$

where $\boldsymbol{g}_\alpha = \boldsymbol{g}(t_{1+\alpha})$ and $t_{1+\alpha} = t_0 + (1+\alpha)\,h$. Note that Newmark's method is recovered with $\alpha = 0$. To obtain an unconditionally stable method of second order, the following constraints must be obeyed:

$$-1/3 \leq \alpha \leq 0 \qquad \theta_1 = \frac{1 - 2\,\alpha}{2} \qquad \theta_2 = \frac{(1-\alpha)^2}{2}$$

The smaller the value of α, the more damping is induced in the numerical solution.

Substituting now:

$$\dot{\boldsymbol{v}}_1 = \alpha_1\,\boldsymbol{u}_1 - \boldsymbol{p} \qquad \text{and} \qquad \boldsymbol{v}_1 = \alpha_4\,\boldsymbol{u}_1 - \boldsymbol{q}$$

Then:

$$\boldsymbol{M}\cdot(\alpha_1\,\boldsymbol{u}_1 - \boldsymbol{p}) + (1+\alpha)\,\boldsymbol{C}\cdot(\alpha_4\,\boldsymbol{u}_1 - \boldsymbol{q}) + (1+\alpha)\,\boldsymbol{K}\,\boldsymbol{u}_1 = \boldsymbol{g}_\alpha + \alpha\,\boldsymbol{C}\,\boldsymbol{v}_0 + \alpha\,\boldsymbol{K}\,\boldsymbol{u}_0$$

And $\boldsymbol{u}_1$ can be found as follows:

$$[\alpha_1\,\boldsymbol{M} + (1+\alpha)\,\alpha_4\,\boldsymbol{C} + (1+\alpha)\,\boldsymbol{K}]\;\boldsymbol{u}_1 = \boldsymbol{g}_\alpha + \boldsymbol{M}\,\boldsymbol{p} + \boldsymbol{C}\;[(1+\alpha)\,\boldsymbol{q} + \alpha\,\boldsymbol{v}_0] + \alpha\,\boldsymbol{K}\,\boldsymbol{u}_0$$

or

$$\boxed{\boldsymbol{u}_1 = (\alpha_1\,\boldsymbol{M} + \alpha_7\,\boldsymbol{C} + \alpha_8\,\boldsymbol{K})^{-1}(\boldsymbol{g}_\alpha + \boldsymbol{M}\,\boldsymbol{p} + \boldsymbol{C}\,\bar{\boldsymbol{q}} + \alpha\,\boldsymbol{K}\,\boldsymbol{u}_0)}$$

with:

$$\boxed{\alpha_7 = (1+\alpha)\,\alpha_4 \qquad \alpha_8 = (1+\alpha) \qquad \bar{\boldsymbol{q}} = \alpha_8\,\boldsymbol{q} + \alpha\,\boldsymbol{v}_0}$$

The FEM discretization of the dynamic problem leads to:

$$\boldsymbol{M}\ddot{\boldsymbol{U}} + \boldsymbol{C}\dot{\boldsymbol{U}} + \boldsymbol{K}\boldsymbol{U} = \boldsymbol{F}$$

which can also be viewed as two first order differential equations in time:

$$\dot{\boldsymbol{U}} = \boldsymbol{V}$$

$$\boldsymbol{M}\dot{\boldsymbol{V}} + \boldsymbol{C}\boldsymbol{V} + \boldsymbol{K}\boldsymbol{U} = \boldsymbol{F}$$

However, to obtain a numerical solution, the prescribed boundary conditions have to be incorporated into the system of equations:

$$\begin{bmatrix}\boldsymbol{M}_{11} & \boldsymbol{M}_{12}\\ \boldsymbol{M}_{21} & \boldsymbol{M}_{22}\end{bmatrix}\begin{Bmatrix}\dot{\boldsymbol{V}}_1\\ \dot{\boldsymbol{V}}_2\end{Bmatrix} + \begin{bmatrix}\boldsymbol{C}_{11} & \boldsymbol{C}_{12}\\ \boldsymbol{C}_{21} & \boldsymbol{C}_{22}\end{bmatrix}\begin{Bmatrix}\boldsymbol{V}_1\\ \boldsymbol{V}_2\end{Bmatrix} + \begin{bmatrix}\boldsymbol{K}_{11} & \boldsymbol{K}_{12}\\ \boldsymbol{K}_{21} & \boldsymbol{K}_{22}\end{bmatrix}\begin{Bmatrix}\boldsymbol{U}_1\\ \boldsymbol{U}_2\end{Bmatrix} = \begin{Bmatrix}\boldsymbol{F}_1\\ \boldsymbol{F}_2\end{Bmatrix}$$

i.e.:

$$\boldsymbol{M}_{11}\dot{\boldsymbol{V}}_1 + \boldsymbol{C}_{11}\boldsymbol{V}_1 + \boldsymbol{K}_{11}\boldsymbol{U}_1 = \boldsymbol{F}_1 - \boldsymbol{M}_{12}\dot{\boldsymbol{V}}_2 - \boldsymbol{C}_{12}\boldsymbol{V}_2 - \boldsymbol{K}_{12}\boldsymbol{U}_2$$

Application of the Newmark's solution to FEM equations

Mesh Generation
Modelling
Post-processing
Eigenvalues
Time: 1st order
Time: 2nd order
Newmark 1
Newmark 2
Newmark 3
Newmark 4
Newmark 5
Newmark 6
Newmark 7
HHT 1
HHT 2
FEM equations 1
▷ FEM equations 2
Rayleigh 1
Examples 1st o.
Examples 2nd o.

Thus:

$$\underbrace{\boldsymbol{M}_{11}}_{\boldsymbol{M}}\underbrace{\dot{\boldsymbol{V}}_1}_{\dot{\boldsymbol{v}}}+\underbrace{\boldsymbol{C}_{11}}_{\boldsymbol{C}}\underbrace{\boldsymbol{V}_1}_{\boldsymbol{v}}+\underbrace{\boldsymbol{K}_{11}}_{\boldsymbol{K}}\underbrace{\boldsymbol{U}_1}_{\boldsymbol{u}}=\underbrace{\boldsymbol{G}}_{\boldsymbol{g}}$$

with:

$$\boldsymbol{G}=\boldsymbol{F_1}-\boldsymbol{M}_{12}\dot{\boldsymbol{V}}_2-\boldsymbol{C}_{12}\boldsymbol{V_2}-\boldsymbol{K}_{12}\boldsymbol{U_2}$$

Or, simply:

$$\boldsymbol{M}\dot{\boldsymbol{v}}+\boldsymbol{C}\boldsymbol{v}+\boldsymbol{K}\boldsymbol{u}=\boldsymbol{g}$$

where the context has to be understood: in this case the equations after the specification of boundary conditions are being considered.

Rayleigh (artificial) damping

Mesh Generation
Modelling
Post-processing
Eigenvalues
Time: 1st order
Time: 2nd order
Newmark 1
Newmark 2
Newmark 3
Newmark 4
Newmark 5
Newmark 6
Newmark 7
HHT 1
HHT 2
FEM equations 1
FEM equations 2
▷ Rayleigh 1
Examples 1st o.
Examples 2nd o.

The dynamics equation derived using the finite element method for the case when no damping exists is given by:

$$\boldsymbol{M}\boldsymbol{A}+\boldsymbol{K}\boldsymbol{U}=\boldsymbol{F}$$

i.e., the $\boldsymbol{C}$ matrix is zero.

Sometimes it is convenient to introduce a little of *artificial damping* in order to make the simulation to converge to the steady state more easily. The so-called Rayleigh damping method can then be adopted. In this method, the $\boldsymbol{C}$ matrix is a weighted addition of the $\boldsymbol{M}$ and $\boldsymbol{K}$ matrices as follows:

$$\underbrace{\boldsymbol{C}}_{\text{Rayleigh}}=ray_M\,\boldsymbol{M}+ray_K\,\boldsymbol{K}$$

where the ray_M and ray_K coefficients have no physical meaning and are usually discovered by *trial-and-error*.

Hence, the full form of the dynamics governing equation has to be solved:

$$\boldsymbol{M}\boldsymbol{A}+\boldsymbol{C}\boldsymbol{V}+\boldsymbol{K}\boldsymbol{U}=\boldsymbol{F}$$

Examples: First Order Problems

Post-processing: the `get_itouts` function

To find the indices corresponding to a particular series of output times, the code `iout=int(tout/dt)` could be used. However, the following code is a better way to find these indices, especially when using arbitrary timesteps:

```
from util import get_itout # auxiliary function
                           # to get indices and times
. . .

# sorted nodes @ the bottom edge
b_ids = m.get_verts(-100, xsorted=True) # ids
b_X   = [m.V[vid][2] for vid in b_ids]   # coordinates

. . .

# plot solution for bottom nodes and some time outputs
touts = [0.005, 0.1, 0.2, 0.5, 1.0]
ITout = get_itout(s.Tout, touts, tol=dt/2.0)
for iout, tout in ITout:
    Ufem = [s.Uout['u'][vid][iout] for vid in b_ids]
    plot(b_X, Ufem)

. . .
```

Diffusion equation: example 1

This example solves the following 1D diffusion equation with the finite element method:

$$\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} = 0 \qquad 0 \le x \le 4$$

for the following boundary conditions:

$$u(x=0) = 1 \quad \text{and} \quad \left.\frac{\partial u}{\partial x}\right|_{x=4} = 0$$

This represents the heating up of a bar with constant temperature at the left end. After a number of time steps, the temperature should reach a constant value of 1 throughout the bar.

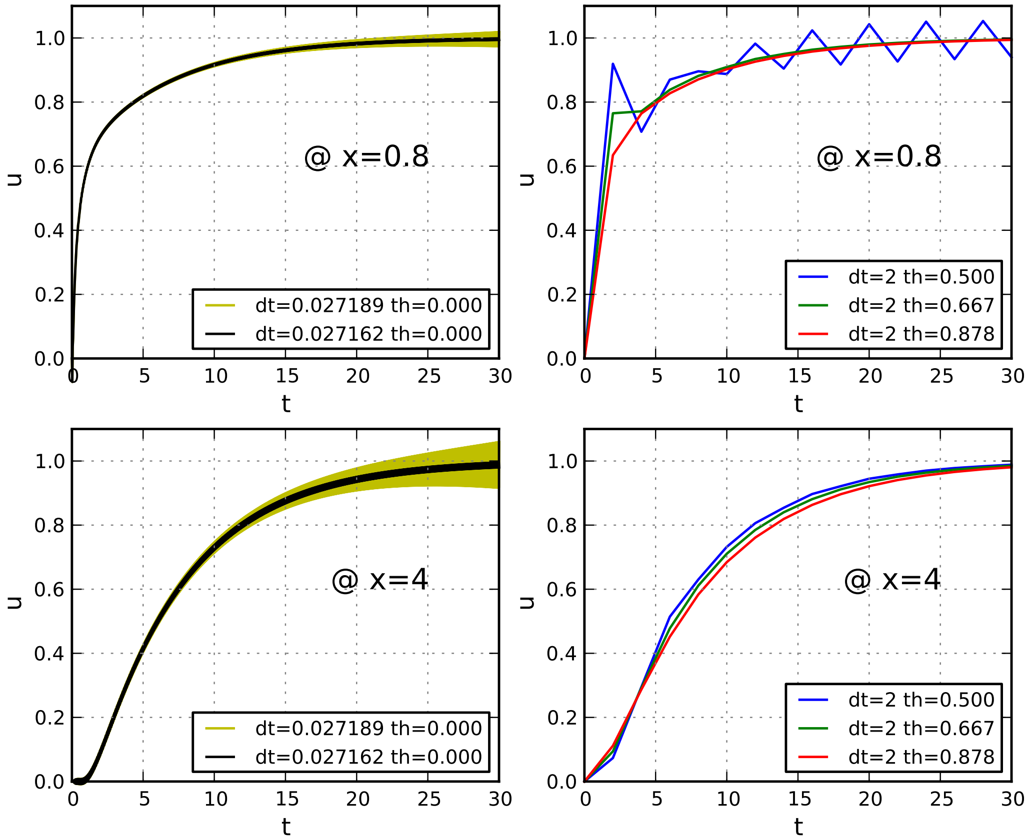
The transient analysis is carried out after the largest eigenvalue is found first in order to find out the critical time-step for an explicit method with $\theta = 0$. It is then shown that for very large time steps, some implicit methods may suffer of small instabilities. Nonetheless, these are due to difficulties on introducing the right initial conditions.

Diffusion equation: example 1 – Python script

oned_diffusion.py

```python
from FEMsolver import * # FEM solver
from FEMmesh   import * # FEM mesh
from msys_fig  import * # auxiliary plotting routines

# set size of figure for presentation
SetForEps(0.8, 500)

# mesh
# ==========================================================
m = Gen1Dmesh(4.,11)
vids = [2, 10]    # vertices for output
nv   = len(vids) # number of vertices for output

# solver and boundary conditions
# ==========================================================
p  = {-1:{'rho':1.0, 'kx':1.0}}
s  = FEMsolver(m, 'Ediffusion1D', p)
vb = {-101:{'u':1.0}}
s.set_bcs(vb=vb)

# eigenvalue analysis
# ==========================================================
s.solve_eigen(dyn=False, dense=True)
lmax   = max(s.L)    # largest lambda
dtcrit = 2.0 / lmax # critical time-step for theta=0
print 'lmax   =', lmax
print 'dtcrit =', dtcrit
#
# solve transient problem with theta=0 for two dt's
# ==========================================================
t0, U0, tf = 0.0, zeros(s.neqs), 30.0
clrs = ['y', 'k'] # colors
for i, dt in enumerate([dtcrit*1.001, dtcrit]):
    s.solve_transient(t0, U0, tf, dt, theta=0.0)
    for k, vid in enumerate(vids):  # for out vertices
        iplt = k*2 + 1              # index of sub-plot
        subplot(nv, 2, iplt)        # one plot for each v
        plot(s.Tout, s.Uout['u'][vid],
             color=clrs[i], clip_on=0,
             label='dt=%.6f th=%.3f'%(dt,0.0))

# solve transient problem with a number of theta's
# ==========================================================
dt = 2.0 # very large time-step
for th in [0.5, 2./3., 0.878]: # for a number of theta's
    s.solve_transient(t0, U0, tf, dt, theta=th)
    for k, vid in enumerate(vids): # for out vertices
        iplt = k*2 + 2             # index of sub-plot
        subplot(nv, 2, iplt)       # one plot for each v
        plot(s.Tout, s.Uout['u'][vid], clip_on=0,
             label='dt=%g th=%.3f'%(dt,th))

# set labels and legends and show figure
# ==========================================================
for k, vid in enumerate(vids): # for output vertices
    xc = m.V[vid][2]           # x-coord of vertex
    ia = k*2 + 1               # index of odd columns
    ib = k*2 + 2               # index of even columns
    for iplt in [ia,ib]:       # for each splot column
        subplot(nv, 2, iplt)   # one plot for each v
        text(25., 0.6, '@ x=%g'%xc, ha='right')
```

Diffusion equation: example 1 – Results



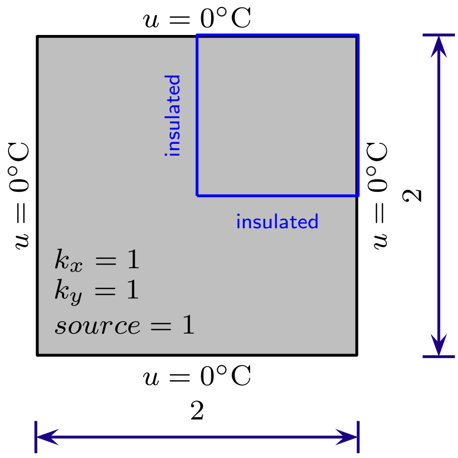
Diffusion equation: example 2 from [Reddy 2006, p494-498]

The plate shown to the right has fixed zero temperature along its borders while its interior has a constant heat source equal to one. Initially, the temperature is continuously zero within the plate.

Although the steady state can be easily calculated, sometimes the transient behaviour has to be understood. Especially when questions such as *How long does it take to achieve equilibrium?* must be answered.

Before a transient analysis, an eigenvalue analysis can be carried out in order to understand the *natural* behaviour of the heat diffusion within this plate, in addition to find the largest eigenvalue limiting the critical time-step for explicit methods.

Note that thanks to symmetry, only one quarter of the domain needs to be considered



Later, a transient analysis can be conducted (e.g. explicit with $\theta = 0$)

Diffusion equation: example 2

The solution of this problem can be easily achieved with the finite element method. Using `FEMsolver`, three methods can be used: `solve_eigen`, `solve_transient`, and later, for comparison, `solve_steady`.

To compare a simulation with $\theta = 0$ (explicit) with a stable one with $\theta = 0.5$, the maximum eigenvalue is found first, allowing the calculation of the critical time step. Then the two transient analyses can be carried out, one after another. Finally, the steady state solution, including its closed-form solution can be plotted.

Two set of data is post-processed: (a) one plot with the variation of the temperature `u` along the lower edge of the one-quarter mesh – i.e. the middle-to-right horizontal section of the plate – for a number of output times; and (b) the variation of the temperature of a node at the centre of the plate from the initial to the final time.

It is expected that the temperature at the centre increases with time due to the source term, achieving the steady state.

Diffusion equation: example 2 – Python script

Mesh Generation
Modelling
Post-processing
Eigenvalues
Time: 1st order
Time: 2nd order
Examples 1st o.
Post-processing
Diffusion ex 1a
Diffusion ex 1b
Diffusion ex 1c
Diffusion ex 2a
Diffusion ex 2b
▷ Diffusion ex 2b
Diffusion ex 2c
Diffusion ex 2d
Diffusion ex 2e
Examples 2nd o.

`reddy_diffusion_eigtrans.py`

```python
import sys              # access system parameters
from FEMsolver import * # FEM solver
from FEMmesh   import * # FEM mesh
from msys_fig  import * # auxiliary plotting routines
from util      import * # auxiliary functions

# input
rotated  = int(sys.argv[1]) if len(sys.argv)>1 else False
showmesh = int(sys.argv[2]) if len(sys.argv)>2 else False

# mesh
# ==============================================================
l  = linspace(0.0, 1.0, 5)
m  = Gen2Dregion(l, l, rotated=rotated)
m.tag_verts_on_line(-100, 0.0, 0.0, 0.0)
m.tag_vert(-101, 0.0, 0.0)
if showmesh:
    SetForEps(1, 330) # set presentation figure size
    m.draw()
    axis([-0.1,1.1,-0.1,1.1])
    Save('reddy_diffusion_eigtrans_rt%d_mesh.eps'%rotated)
    sys.exit(0) # just show mesh and then exit
else:
    SetForEps(1.5, 330) # set presentation figure size

# node @ (x=0,y=0)
n00 = m.get_verts(-101)[0] # assuming it's the first one

# sorted nodes @ the bottom edge
b_ids = m.get_verts(-100, xsorted=True) # ids
b_ids.insert(0, n00)                    # prepend n00
b_X   = [m.V[vid][2] for vid in b_ids]  # coordinates
#
# solver and boundary conditions
# ==============================================================
rho, kx, ky, src = 1.0, 1.0, 1.0, 1.0
p = {-1:{'rho':rho, 'kx':kx, 'ky':ky, 'source':src}}
s = FEMsolver(m, 'EdiffusionTri', p)

# boundary conditions
eb = {-11:{'u':0.0}, -12:{'u':0.0}}
s.set_bcs(eb=eb)

# eigenvalue analysis
# ==============================================================
s.solve_eigen(dyn=False, dense=True)
print 'omegas =', sqrt(s.L)

# critical time-step for theta=0
lmax   = max(s.L) # largest lambda
dtcrit = 2.0 / lmax
print 'lmax   =', lmax
print 'dtcrit =', dtcrit

# solve transient problem with theta=0
# ==============================================================
U0 = zeros(s.neqs)
dt = dtcrit * 1.015 # using a 1.5% larger dt
t0, tf, th = 0.0, 1.0, 0.0
s.solve_transient(t0, U0, tf, dt, theta=th)

# plot transient sol for all points and some time outputs
subplot(2,1,1)
touts = [0.005, 0.1, 0.2, 0.5, 1.0]
ITout = get_itout(s.Tout, touts, dt/2.0)
clrs  = ['c','b','m','#f99a04','r']
k     = 0 # index to select color
```

Diffusion equation: example 2 – Python script

Mesh Generation
Modelling
Post-processing
Eigenvalues
Time: 1st order
Time: 2nd order
Examples 1st o.
Post-processing
Diffusion ex 1a
Diffusion ex 1b
Diffusion ex 1c
Diffusion ex 2a
Diffusion ex 2b
Diffusion ex 2b
▷ Diffusion ex 2c
Diffusion ex 2d
Diffusion ex 2e
Examples 2nd o.

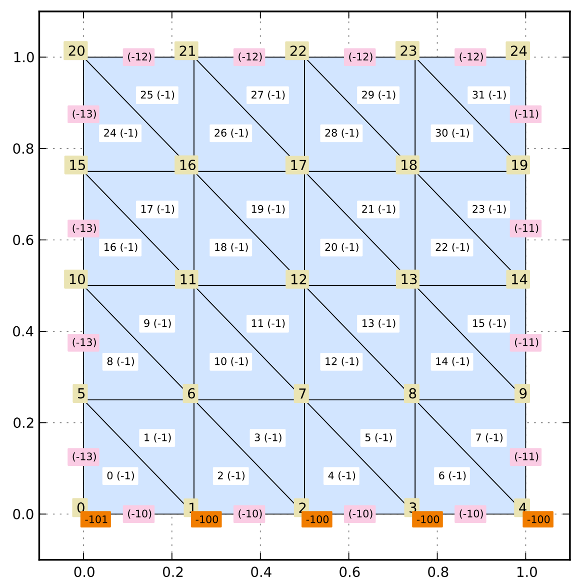
(continued)

```
for iout, tout in ITout:
    Ufem = [s.Uout['u'][vid][iout] for vid in b_ids]
    plot(b_X, Ufem, '-', color=clrs[k],
            label='th=%g t=%.1f'%(th,tout))
    k += 1 # next color

# plot the transient solution for a point located @ (0,0)
subplot(2,1,2)
plot(s.Tout, s.Uout['u'][n00], 'r-',
      label='node # %d: th=%g dt=%g'%(n00,th,dt))

# solve transient problem with theta=0.5
# ==============================================================
th = 0.5
s.solve_transient(t0, U0, tf, dt, theta=th)

# plot transient solution again for all points and touts
subplot(2,1,1)
ITout = get_itout(s.Tout, touts, tol=dt/2.0)
k     = 0 # index to pick color
for iout, tout in ITout:
    Ufem = [s.Uout['u'][vid][iout] for vid in b_ids]
    plot(b_X, Ufem, '+-', color=clrs[k],
            label='th=%g t=%.1f'%(th,tout))
    k += 1 # next color

# plot again the transient solution for a point @ (0,0)
subplot(2,1,2)
plot(s.Tout, s.Uout['u'][n00], 'b.', zorder=-1,
      label='node # %d: th=%g dt=%g'%(n00,th,dt))
#
# steady state solution: numerical and analytical
# ==============================================================
def usteady(x,y): # analytical solution
    res = 0.0
    for i in range(1,41):
        a = 0.5*pi*(2.*i-1.)
        res += ((-1.)**float(i))*cos(a*y)*cosh(a*x) /   \
                 ((a**3.) * cosh(a))
    return (0.5*src/kx) * ((1.-y**2.) + 4.0*res)

# solve steady problem
s.solve_steady()
Unum = s.get_u()

# plot steady solution for all points @ equilibrium state
subplot(2,1,1)
xsol = linspace(0.0, 1.0, 101)
plot(xsol, usteady(xsol,0.0), 'g-', lw=2,
    label='sol: steady')
plot(b_X, Unum['u'][b_ids], 'yo', ls=':',
     label='fem: steady')

# set labels and legends and show figure
# ==============================================================
subplot(2,1,1)
lg1 = Gll('x', 'u @ (x,y=0)', leg_out=True, leg_ncol=3)
subplot(2,1,2)
x0, y0 = m.V[n00][2], m.V[n00][3]
Gll('t', 'u @ (x=%g,y=%g)'%(x0,y0), leg_loc='lower right')
axis([0.,1.,0.,0.35])
Save('reddy_diffusion_eigtrans_rt%d.eps'%rotated, lg1)
```

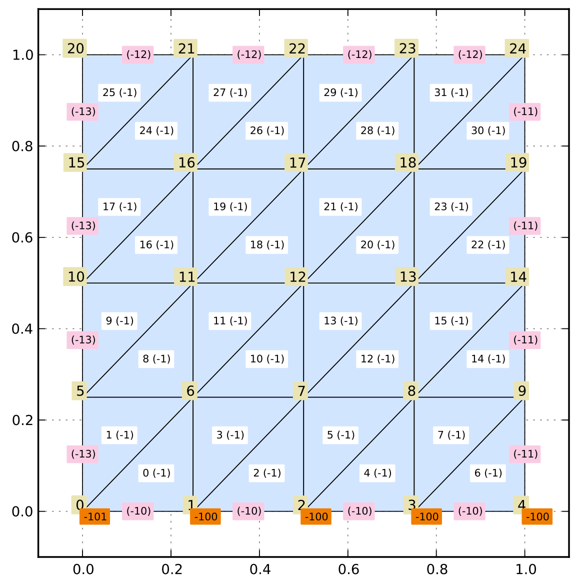

Diffusion equation: example 2 – Meshes

Mesh Generation
Modelling
Post-processing
Eigenvalues
Time: 1st order
Time: 2nd order
Examples 1st o.
Post-processing
Diffusion ex 1a
Diffusion ex 1b
Diffusion ex 1c
Diffusion ex 2a
Diffusion ex 2b
Diffusion ex 2b
Diffusion ex 2c
▷ Diffusion ex 2d
Diffusion ex 2e
Examples 2nd o.

standard mesh

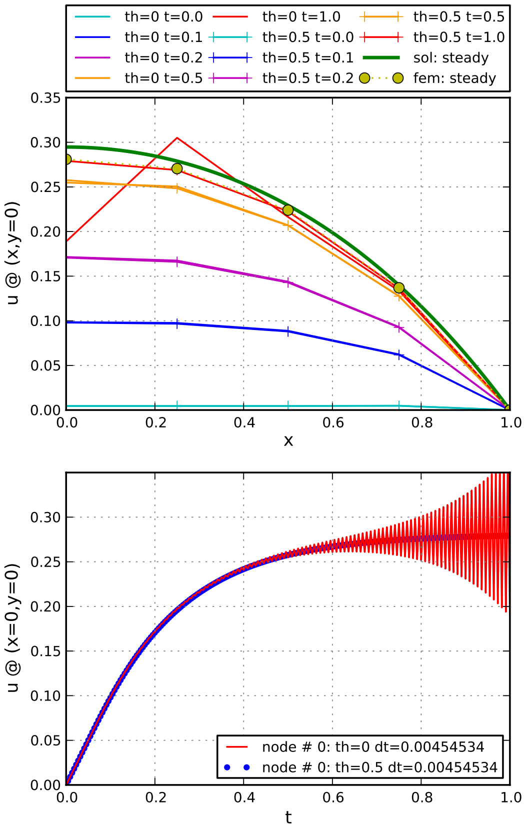


mesh with *rotated* triangles

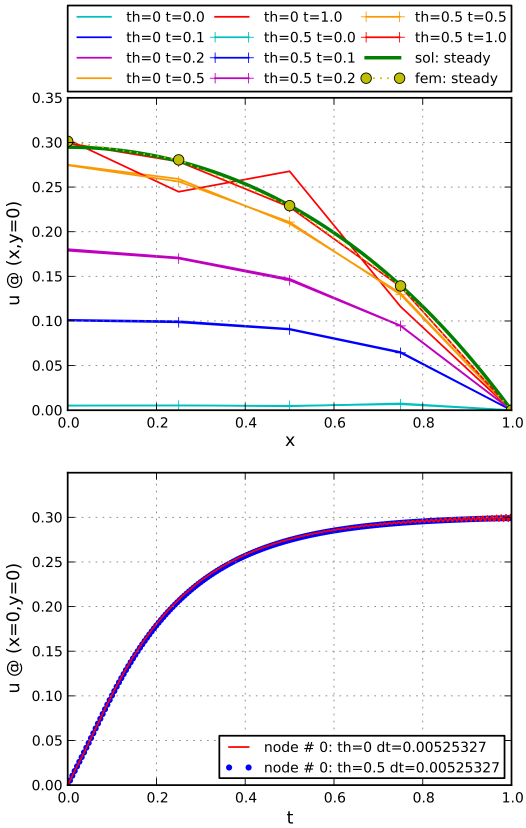


Diffusion equation: example 2 – Results

standard mesh



mesh with *rotated* triangles



Examples: Second Order Problems

Longitudinal vibration of a rod

This example solves the longitudinal vibration of a rod fixed at its left end. The `EelasticRod` is used even though it is a 2D element. To do so, all vertical displacements are prevented.

After an eigenvalue analysis is run, the transient analysis is carried out with `solve_dynamics`. The largest eigenvalue is therefore found in order to calculate the critical time step for a choice of Newmark's parameters such that $\theta_1 = 0.5$ and $\theta_2 = 0.05$.

The dynamics problem corresponds to a forced vibration due to a horizontal force equal to $\sin(\pi\, t/2)$ applied to the right end of the bar.

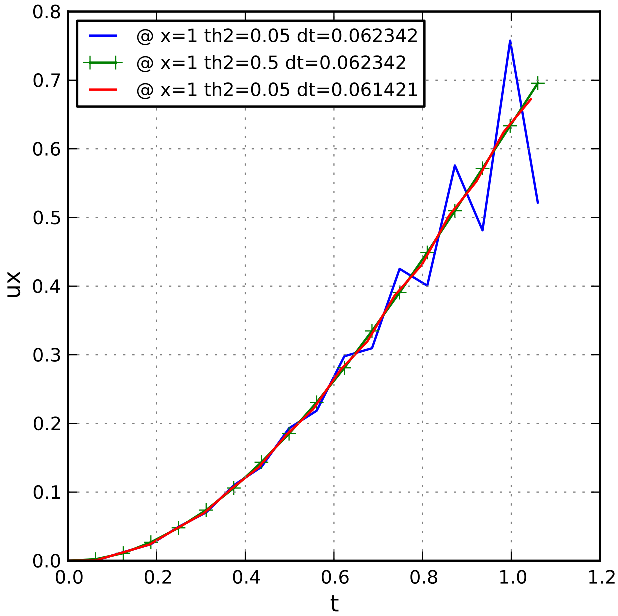
Three dynamics solution are found for a combination of θ_1, θ_2 and the time step. These include a simulation using a time step slightly larger that the critical one in order to illustrated instability.

Longitudinal vibration of a rod – Python code

`oned_rod_vibration.py`

```python
from FEMsolver import * # FEM solver
from FEMmesh   import * # FEM mesh
from msys_fig  import * # auxiliary plotting routines
SetForEps(1, 300) # set size of presentation figure

# mesh
# ======================================================
Lx, nx = 1.0, 11
dx     = Lx / float(nx-1)
V = [[i, -100, i*dx, 0.] for i in range(nx)]
C = [[i,   -1, [i, i+1]] for i in range(nx-1)]
V[ 0][1] = -101;  V[-1][1] = -102
m = FEMmesh(V, C)
vid = m.get_verts(-102)[0] # node @ the right-end

# solver and boundary conditions
# ======================================================
p  = {-1:{'rho':1.0, 'E':1.0, 'A':1.0}}
vb = {-101:{'ux':0.0, 'uy':0.0}, -100:{'uy':0.0},
      -102:{'uy':0.0, 'fx':lambda t:sin(pi*t/2.0)}}
s  = FEMsolver(m, 'EelasticRod', p)
s.set_bcs(vb=vb)

# eigenvalue analysis
# ======================================================
s.solve_eigen(dense=True, evecs=True) # with eigenvectors
s.write_vtu('oned_rod_vibration_modes') # draw shape-modes

# critical time-step corresponding to Newmark parameters
th1, th2 = 0.5, 0.05  # Newmark parameters
lmax     = max(s.L)    # largest lambda
om_max   = sqrt(lmax) # largest natural frequency
dtcrit   = sqrt(2.0)/(om_max*sqrt(th1-th2)) # critical dt
print 'lmax   =', lmax
print 'om_max =', om_max
print 'dtcrit =', dtcrit

# solve dynamics problem with th2 = 0.05 and larger dtcrit
# ======================================================
U0 = zeros(s.neqs)  # initial displacements
V0 = zeros(s.neqs)  # initial velocities
t0, tf = 0.0, 1.0   # initial and final time
dt = dtcrit * 1.015 # larger than critical time-step
s.solve_dynamics(t0,U0,V0,tf,dt, theta1=th1, theta2=th2)
# plot solution @ vid
plot(s.Tout, s.Uout['ux'][vid],
     label='@ x=%g th2=%g dt=%.6f'%(Lx,th2,dt))

# solve dynamics problem with th2 = 0.5 and larger dtcrit
# ======================================================
th2 = 0.5
s.solve_dynamics(t0,U0,V0,tf,dt, theta1=th1, theta2=th2)
# plot solution @ vid
plot(s.Tout, s.Uout['ux'][vid], marker='+',
     label='@ x=%g th2=%g dt=%.6f'%(Lx,th2,dt))

# solve dynamics problem with th2 = 0.05 and dtcrit
# ======================================================
th2, dt = 0.05, dtcrit
s.solve_dynamics(t0,U0,V0,tf,dt, theta1=th1, theta2=th2)
# plot solution @ vid
plot(s.Tout, s.Uout['ux'][vid],
     label='@ x=%g th2=%g dt=%.6f'%(Lx,th2,dt))

# set labels and legends and show figure
# ======================================================
Gll('t', 'ux')
Save('oned_rod_vibration.eps')
```

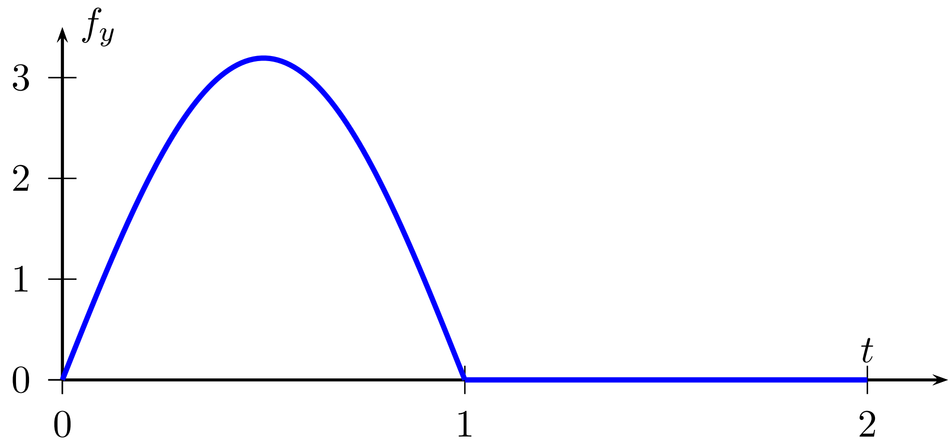
The mode shapes can be drawn in ParaView. The evolution of the horizontal displacement of a node at the right end of the bar is given below.



A vertical force is applied at the right end of a cantilever beam of unit length. The magnitude of this force is given by (see figure below):

$$f_y(t) = \begin{cases} 3.194 \sin(\pi\, t) & \text{if} \quad t < 1 \\ 0 & \text{otherwise} \end{cases}$$

Find the transient response (dynamics) for $0 \leq t \leq 1.8$. The beam parameters are: $E = 3.194$, $A = 1$, $I = 1$ and $\rho = 1$.



Transversal vibration of a beam – Python code

Mesh Generation
Modelling
Post-processing
Eigenvalues
Time: 1st order
Time: 2nd order
Examples 1st o.
Examples 2nd o.
Rod 1a
Rod 1b
Rod 1c
Beam 1a
▷ Beam 1b
Beam 1c
Beam 1d
Elasticity 1a
Elasticity 1b
Elasticity 1c
Elasticity 1d
Elasticity 1e

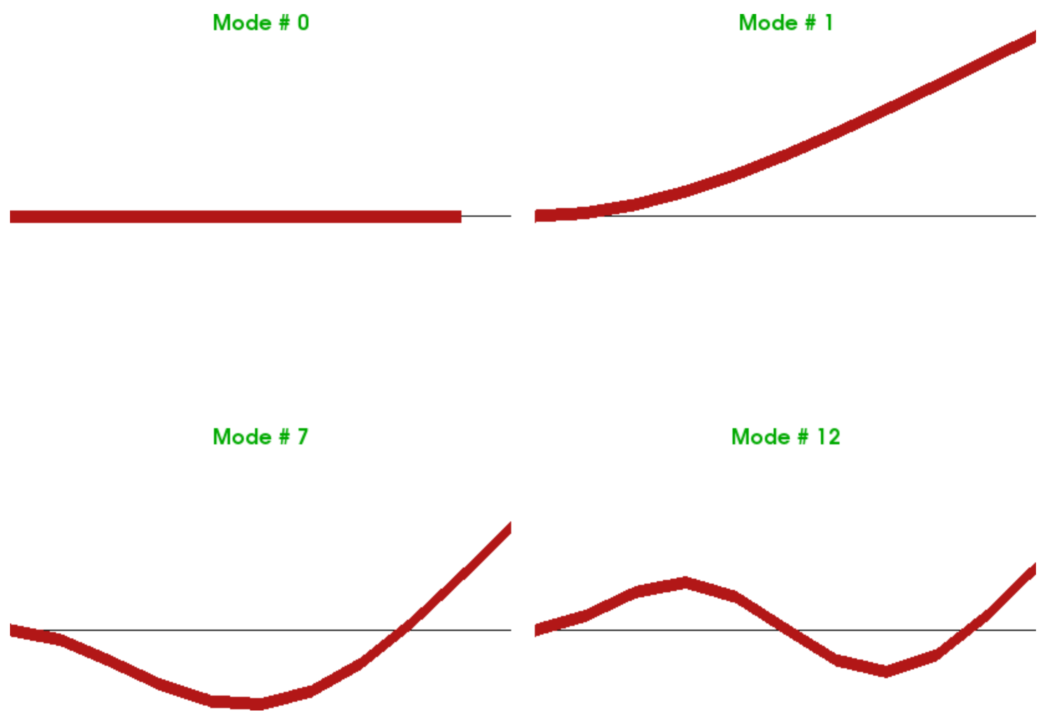
beam_vibration.py

```
from FEMsolver import * # FEM solver
from FEMmesh   import * # FEM mesh
from msys_fig  import * # auxiliary plotting routines
SetForEps(1, 300) # set size of presentation figure

# mesh
# ==========================================================
Lx, nx = 1.0, 11
dx     = Lx / float(nx-1)
V = [[i,    0, i*dx, 0.] for i in range(nx)]
C = [[i,   -1, [i, i+1]] for i in range(nx-1)]
V[ 0][1] = -101;  V[-1][1] = -102
m = FEMmesh(V, C)
vid = m.get_verts(-102)[0] # node @ the right-end

# solver and boundary conditions
# ==========================================================
p  = {-1:{'rho':1.0, 'E':3.194, 'A':1.0, 'I':1.0}}
vb = {-101:{'ux':0.0, 'uy':0.0, 'wz':0.0},
      -102:{'fy':lambda t:3.194*sin(pi*t) if t<1. else 0.}}
s  = FEMsolver(m, 'EelasticBeam', p)
s.set_bcs(vb=vb)

# eigenvalue analysis
# ==========================================================
s.solve_eigen(dense=True, evecs=True) # with eigenvectors
s.write_vtu('beam_vibration_modes') # shape-modes

# critical time-step corresponding to Newmark parameters
th1, th2 = 0.5, 0.05   # Newmark parameters
lmax     = max(s.L)    # largest lambda
om_max   = sqrt(lmax)  # largest natural frequency
dtcrit   = sqrt(2.0)/(om_max*sqrt(th1-th2)) # critical dt
print 'lmax   =', lmax
print 'om_max =', om_max
print 'dtcrit =', dtcrit # <<<<<<<<<<<<<<<<<<<<<< too small

# solve dynamics problem with th2=0.5
# ==========================================================
U0 = zeros(s.neqs)   # initial displacements
V0 = zeros(s.neqs)   # initial velocities
t0, tf = 0.0, 1.8    # initial and final time
dt     = 0.05        # larger time-step
th2    = 0.5         # for unconditional stability
s.solve_dynamics(t0,U0,V0,tf,dt, theta1=th1, theta2=th2)
plot(s.Tout, s.Uout['uy'][vid], 'r.',
     label='fem @ x=%g th2=%g dt=%g'%(Lx,th2,dt))

# plot analytical solution
# ==========================================================
def uysol(t):
    if t<1.0: return 0.441*sin(pi*t)-0.216*sin(2.0*pi*t)
    return -0.432*sin(6.284*(t-1.0))
Uysol = vectorize(uysol)
T = linspace(t0, tf, 101)
plot(T, Uysol(T), 'g-', label='solution', zorder=-1)

# set labels and legends and show figure
# ==========================================================
Gll('t', 'uy')
axis([0.,1.8,-0.6,0.6])
Save('beam_vibration.eps')
```

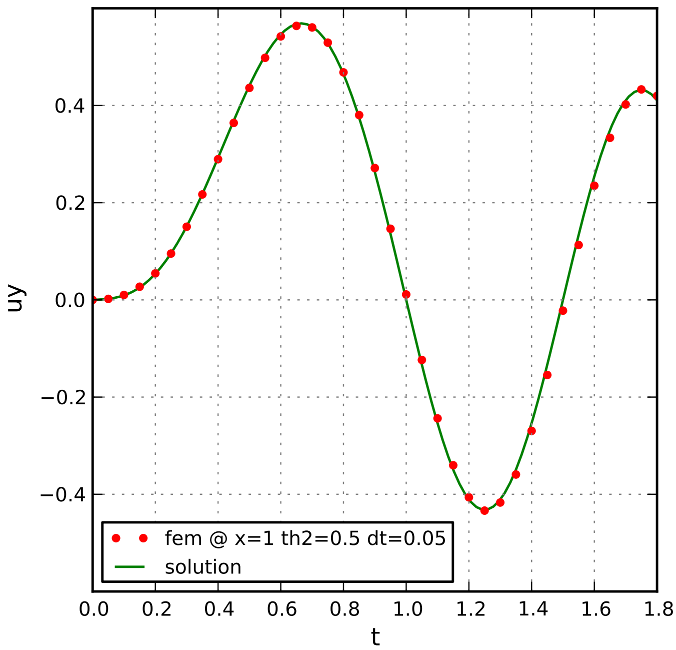

Transversal vibration of a beam – Python code

Mesh Generation
Modelling
Post-processing
Eigenvalues
Time: 1st order
Time: 2nd order
Examples 1st o.
Examples 2nd o.
Rod 1a
Rod 1b
Rod 1c
Beam 1a
Beam 1b
▷ Beam 1c
Beam 1d
Elasticity 1a
Elasticity 1b
Elasticity 1c
Elasticity 1d
Elasticity 1e

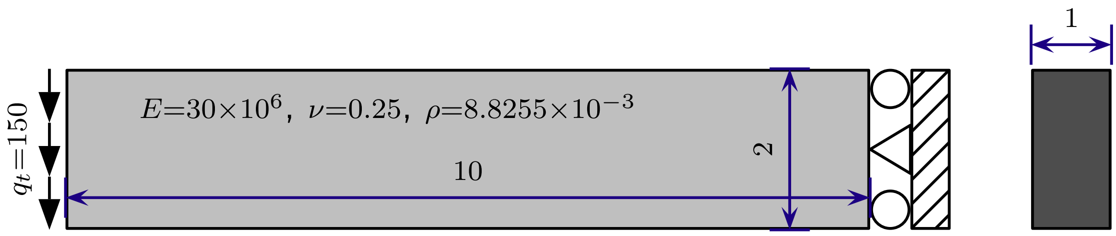
The mode shapes can be animated in ParaView. A screeenshot with four modes is shown below.



The evolution of the vertical displacement of a node at the right end of the beam is shown below.



To investigate the 2D stress distribution within a linear elastic cantilever beam, a finite element simulation using solid elements can be carried out. The figure below shows the geometry considered in this example.



Note that all nodes at the right-hand side have only horizontal fixities; except a point at the middle section which has both degrees of freedom fixed. This is a plane-stress problem with unit thickness.

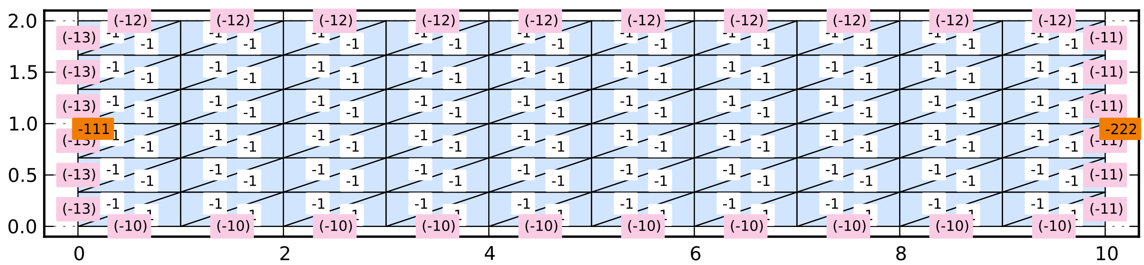
Vibration of cantilever beam – Python code

reddy_cantilever_dyn.py

```python
import sys            # access system parameters
from FEMsolver import * # FEM solver
from FEMmesh   import * # FEM mesh
from msys_fig  import * # auxiliary plotting routines

# input
showmesh = int(sys.argv[1]) if len(sys.argv)>1 else False

# mesh
# ==========================================================
Lx, Ly, thick, nx, ny = 10.0, 2.0, 1.0, 11, 7
xc, yc = linspace(0.0,Lx,nx), linspace(0.0,Ly,ny)
m = Gen2Dregion(xc, yc, rotated=True, vtags=False)
m.tag_vert(-111, 0., Ly/2.) # middle/left node
m.tag_vert(-222, Lx, Ly/2.) # middle/right node
if showmesh:
    SetForEps(0.2, 600) # set presentation figure size
    m.draw(ids=False, ypct=0)
    axis('equal')
    axis([-0.1,10.1,-0.1,2.1])
    Save('reddy_cantilever_dyn_mesh.eps')
    sys.exit(0) # just show mesh and then exit
else:
    SetForEps(1, 300) # set presentation figure size

# selected vertex for output
vid    = m.get_verts(-111)[0]      # vertex id
xc, yc = m.V[vid][2], m.V[vid][3] # coordinates
#
# solver and boundary conditions
# ==========================================================
p  = {-1:{'E':30.e6, 'nu':0.25, 'rho':8.8255e-3,
    'pstress':True, 'thick':thick}}
s  = FEMsolver(m, 'EelasticTri', p)
vb = {-222:{'ux':0.0, 'uy':0.0}}
eb = {-11:{'ux':0.0}, -13:{'qnqt':(0.0, 150.0)}}
s.set_bcs(vb=vb, eb=eb)

# eigenvalue analysis
# ==========================================================
s.solve_eigen(dense=True)

# critical time-step corresponding to Newmark parameters
th1, th2 = 0.5, 1./3. # Newmark parameters
lmax     = max(s.L)    # largest lambda
om_max   = sqrt(lmax)  # largest natural frequency
dtcrit   = sqrt(2.0)/(om_max*sqrt(th1-th2)) # critical dt
print 'lmax   =', lmax
print 'om_max =', om_max
print 'dtcrit =', dtcrit

# solve dynamics problem with th2 = 1/3
# ==========================================================
U0 = zeros(s.neqs)      # initial displacements
V0 = zeros(s.neqs)      # initial velocities
t0, tf = 0.0, 0.005     # initial and final time
dt = dtcrit + 4.0e-10 # larger than critical time-step
s.solve_dynamics(t0,U0,V0,tf,dt, theta1=th1, theta2=th2)

# plot displacement evolution @ selected node
plot(s.Tout, s.Uout['uy'][vid], 'y-', lw=3,
     label='fem: th2=%.3f, dt=%.6e'%(th2,dt))
#
```

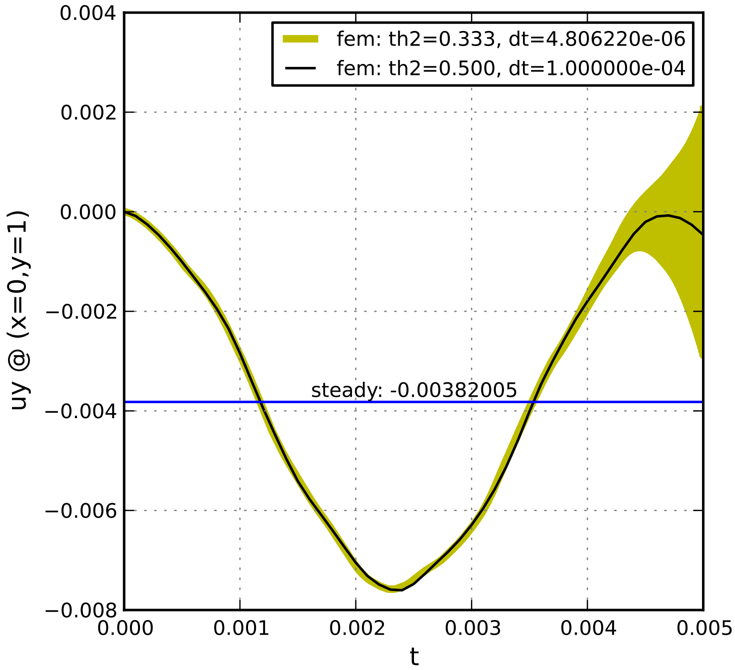

Vibration of cantilever beam – Python code


```python
# solve dynamics problem with th2 = 0.5 and large dt
# ==========================================================
th2, dt = 0.5, 0.0001
s.solve_dynamics(t0,U0,V0,tf,dt, theta1=th1, theta2=th2,
      vtu_fnkey='reddy_cantilever_dyn', # save VTU files
      vtu_dtout=0.0005, # only generate few VTU outputs
      vtu_extrap=True)  # extrapolate second. vals for VTU

# plot displacement evolution @ selected node
plot(s.Tout, s.Uout['uy'][vid], 'k-',
     label='fem: th2=%.3f, dt=%.6e'%(th2,dt))
#
# solve for equilibrium state
# ==========================================================
s.solve_steady()
s.extrapolate()
uystd = s.get_u()['uy'][vid] # steady uy @ selected node
axhline(uystd)
text(0.0025, uystd+0.0001, 'steady: %g'%uystd,
     ha='center', fontsize=8)

# set labels and legend and show figure
# ==========================================================
Gll('t','uy @ (x=%g,y=%g)' % (xc,yc))
axis([0.,0.005,-0.008,0.004])
Save('reddy_cantilever_dyn.eps')
```



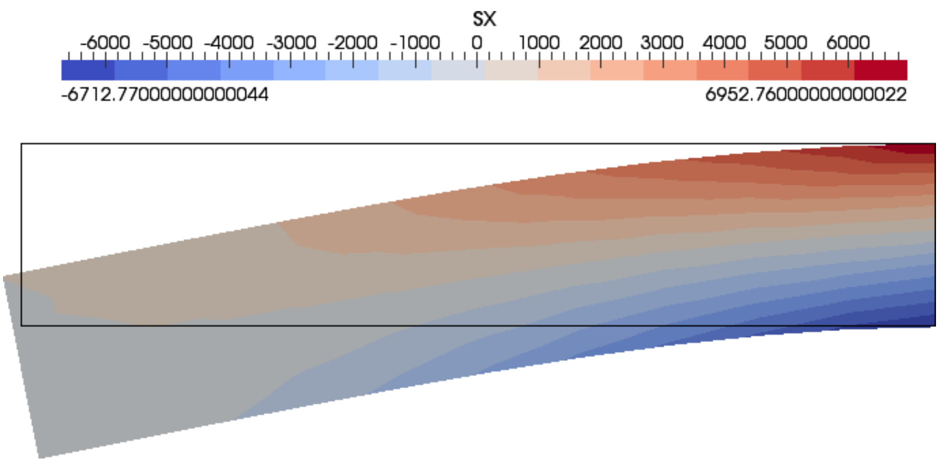
Vibration of cantilever beam – Results

The evolution of the vertical displacement of a node at the left end (middle node with tag -111) is shown below.



Vibration of cantilever beam – Results

The deformed geometry obtained with ParaView is shown below. Colors indicate the horizontal stress component σ_x values. Deformations are magnified by 200x.



Numerical Methods in Engineering – Part VI

Dr Dorival Pedroso

May 28, 2012

Fundamentals

Green-Gauss Theorem

$$
\begin{aligned}
\int_\Omega W \,\operatorname{div} \underset{,}{\boldsymbol{v}} \,\mathrm{d}\Omega &= \int_\Omega \operatorname{div}\,(W\, \underset{,}{\boldsymbol{v}})\,\mathrm{d}\Omega - \int_\Omega \underset{,}{\mathbf{grad}}\, W \bullet \underset{,}{\boldsymbol{v}}\,\mathrm{d}\Omega \\
&= \int_\Gamma W\, \underbrace{\underset{,}{\boldsymbol{v}} \bullet \underset{,}{\hat{\boldsymbol{n}}}}_{\bar{q}}\,\mathrm{d}\Gamma - \int_\Omega \underset{,}{\mathbf{grad}}\, W \bullet \underset{,}{\boldsymbol{v}}\,\mathrm{d}\Omega \\
&= \sum_n W_n \left(\int_\Gamma S_n\, \bar{q}\,\mathrm{d}\Gamma - \int_\Omega \underset{,}{\bar{\boldsymbol{G}}}_n \bullet \underset{,}{\boldsymbol{v}}\,\mathrm{d}\Omega \right)
\end{aligned}
$$

Coupled Systems

Coupled Systems: Porous Media

The mechanical behaviour of porous media depends on whether its porous matrix is filled with fluids or not. A typical example is a fully water saturated soil column subject to some sort of surface loading. If the water within the porous matrix can flow out, then the column will deform; otherwise, all pressure goes to the fluid and the soil skeleton will not deform. The principle of effective stress, which states what part of stresses goes to the soil skeleton, is of paramount importance in describing the mechanical behaviour of this **coupled** system.
Traditionally, the Terzaghi principle of effective stress is considered:

$$\underset{\approx}{\boldsymbol{\sigma}}' = \underset{\approx}{\boldsymbol{\sigma}} + p_w \, \underset{\sim}{\boldsymbol{I}}$$

or

$$\begin{Bmatrix} \sigma'_x \\ \sigma'_y \\ \sigma'_z \\ \sqrt{2}\,\sigma'_{xy} \end{Bmatrix} = \begin{Bmatrix} \sigma_x \\ \sigma_y \\ \sigma_z \\ \sqrt{2}\,\sigma_{xy} \end{Bmatrix} + p_x \begin{Bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{Bmatrix}$$

Coupled Systems: Porous Media

The material model is then developed in such a way that a relation between effective stresses ($\underset{\approx}{\boldsymbol{\sigma}}'$) and strains is represented. This relation is necessary since the mechanical behaviour of the porous skeleton depends primarily on the effective stresses and not on the total stresses ($\underset{\approx}{\boldsymbol{\sigma}}$). For example, in the soil column, if all the pressure goes to the water, the effective stresses are zero and no mechanical work can happen on the soil skeleton.

Although very rarely soils behave elastically, it is insightful to develop a numerical model based on elasticity in order to understand the coupled phenomenon including deformations and matrix-water flow (seepage). Thus, a linear elastic model is considered here (small strains):

$$\underset{\approx}{\boldsymbol{\sigma}}' = \underset{\approx\approx}{\boldsymbol{D}}^e : \underset{\approx}{\boldsymbol{\varepsilon}}$$

or, for 2D problems and considering Mandel's basis:

$$\boldsymbol{\sigma'} = \boldsymbol{D}\,\boldsymbol{\varepsilon}$$

where $\boldsymbol{\sigma'}$ and ε are the two 4×1 vectors show earlier.

The governing balance of momentum system, after FEM space-discretisation, can now be written as follows (since $\boldsymbol{\sigma} = \boldsymbol{\sigma'} - p_w\,\boldsymbol{I}$):

$$\boldsymbol{M}\boldsymbol{A} + \boldsymbol{C}\boldsymbol{V} + \int_\Omega \boldsymbol{B}^T \boldsymbol{\sigma}\,\mathrm{d}\Omega = \boldsymbol{F}$$

$$\boldsymbol{M}\boldsymbol{A} + \boldsymbol{C}\boldsymbol{V} + \int_\Omega \boldsymbol{B}^T \boldsymbol{\sigma'}\,\mathrm{d}\Omega - \int_\Omega p_w\,\boldsymbol{B}^T \boldsymbol{I}\,\mathrm{d}\Omega = \boldsymbol{F}$$

where it is assumed that the assembly process has been applied for the integral expressions. This can now be expressed as:

$$\boxed{\boldsymbol{M}\boldsymbol{A} + \boldsymbol{C}\boldsymbol{V} + \boldsymbol{K}\boldsymbol{U} - \boldsymbol{Q}\boldsymbol{P} = \boldsymbol{F}}$$

with:

$$\boldsymbol{K} = \int_\Omega \boldsymbol{B}^T \boldsymbol{\sigma'}\,\mathrm{d}\Omega \quad \text{and} \quad \boldsymbol{Q} = \int_\Omega \boldsymbol{B}^T \boldsymbol{I}\bar{\boldsymbol{S}}\,\mathrm{d}\Omega$$

after assuming a space approximation of the water pressure according to:

$$p_w = \bar{\boldsymbol{S}}\boldsymbol{P}$$

The key characteristic of the coupled problem is that two governing systems have to solved at the same time. The other system to be solved is the continuity equation for the water phase. In terms of pressure, this is given by:

$$\dot{\varepsilon}_v + C_w\,\dot{p}_w - \frac{\partial w_x}{\partial x} - \frac{\partial w_y}{\partial y} = 0$$

Coupled Systems: Porous Media

$$\boxed{\begin{aligned}
0 &= C_{pw}\,\dot{p}_w + C_{vs}\ \mathrm{div}\,\underset{,}{\boldsymbol{v}} + \mathrm{div}\,(\rho_w\,\underset{,}{\boldsymbol{w}}_w)\\
0 &= \rho^w\,\underset{,}{\boldsymbol{a}} + \rho^w\,\underset{,}{\dot{\boldsymbol{w}}}_w + \rho^w\,\underset{,,}{\boldsymbol{\ell}}_w \bullet \underset{,}{\boldsymbol{w}}_w - \rho^w\,\underset{,}{\boldsymbol{g}} - \underset{,}{\boldsymbol{f}}_{wd} + \underset{,}{\mathbf{grad}}\,p_w\\
0 &= \rho\,\underset{,}{\boldsymbol{a}} + \rho_w\,\underset{,}{\dot{\boldsymbol{w}}}_w + \rho_w\,\underset{,,}{\boldsymbol{\ell}}_w \bullet \underset{,}{\boldsymbol{w}}_w - \rho\,\underset{,}{\boldsymbol{g}} - \underset{,}{\mathbf{div}}\,\underset{,,}{\boldsymbol{\sigma}}
\end{aligned}}$$

$$\boxed{\begin{aligned}
C_{pw} &= \eta\,S_w\,C_w - \eta\,\rho^w\,C_c\\
C_{vs} &= S_w\,\rho^w
\end{aligned}}$$

$$-C_c\,\dot{p}_w = \dot{S}_w \quad \text{and} \quad C_w = \frac{\rho^{wo}}{K_w}$$

$$0 = (\eta\,S_w\,C_w - \eta\,\rho^w\,C_c)\,\dot{p}_w + S_w\,\rho^w\ \mathrm{div}\,\underset{,}{\boldsymbol{v}} + \mathrm{div}\,(\rho_w\,\underset{,}{\boldsymbol{w}}_w)$$

$$0 = \eta\,S_w\,C_w\,\dot{p}_w - \eta\,\rho^w\,C_c\,\dot{p}_w + S_w\,\rho^w\ \mathrm{div}\,\underset{,}{\boldsymbol{v}} + \mathrm{div}\,(\rho_w\,\underset{,}{\boldsymbol{w}}_w)$$

$$0 = \eta\,S_w\,\frac{\rho^w}{K_w}\,\dot{p}_w + \eta\,\rho^w\,\dot{S}_w + S_w\,\rho^w\ \mathrm{div}\,\underset{,}{\boldsymbol{v}} + \mathrm{div}\,(\rho_w\,\underset{,}{\boldsymbol{w}}_w)$$

$$0 = \frac{\eta\,S_w}{K_w}\,\dot{p}_w + \eta\,\dot{S}_w + S_w\ \mathrm{div}\,\underset{,}{\boldsymbol{v}} + \frac{1}{\rho^w}\,\mathrm{div}\,(\rho_w\,\underset{,}{\boldsymbol{w}}_w)$$

Coupled Systems: Porous Media

Balance of linear momentum of water:

$$0 = \rho^w\,\underset{,}{\boldsymbol{a}} - \rho^w\,\underset{,}{\boldsymbol{g}} - \underset{,}{\boldsymbol{f}}_{wd} + \underset{,}{\mathbf{grad}}\,p_w$$

$$\boxed{\underset{,}{\boldsymbol{f}}_{wd} = \underset{,}{\mathbf{grad}}\,p_w + \rho^w\,\underset{,}{\boldsymbol{a}} - \rho^w\,\underset{,}{\boldsymbol{g}}}$$

Drag force:

$$\underset{,}{\boldsymbol{f}}_{wd} = -\frac{\eta\,S_w\,\gamma_w^{ref}}{k_w^r}\,(\underset{,,}{\boldsymbol{k}}_w^{sat})^{-1} \bullet \underset{,}{\boldsymbol{w}}_w$$

$$\underset{,,}{\boldsymbol{k}}_w^{sat} \bullet \underset{,}{\boldsymbol{f}}_{wd} = -\frac{\eta\,S_w\,\gamma_w^{ref}}{k_w^r}\,\underset{,}{\boldsymbol{w}}_w$$

$$\boxed{\eta\,S_w\,\underset{,}{\boldsymbol{w}}_w = -\frac{k_w^r}{\gamma_w^{ref}}\underset{,,}{\boldsymbol{k}}_w^{sat} \bullet \underset{,}{\boldsymbol{f}}_{wd}}$$

Thus, the generalized Darcy's law is:

$$\boxed{\eta\,S_w\,\underset{,}{\boldsymbol{w}}_w = -\frac{k_w^r}{\gamma_w^{ref}}\underset{,,}{\boldsymbol{k}}_w^{sat} \bullet \left(\underset{,}{\mathbf{grad}}\,p_w + \rho^w\,\underset{,}{\boldsymbol{a}} - \rho^w\,\underset{,}{\boldsymbol{g}}\right)}$$

Coupled Systems: Porous Media

For fully water saturated (isotropic medium):

$$
\begin{aligned}
\eta\, \underset{,}{\boldsymbol{w}}_w &= -\frac{1}{\gamma_w^{ref}} \underset{,,}{\boldsymbol{k}}_w^{sat} \bullet \left(\underset{,}{\mathbf{grad}}\, p_w + \rho^w\, \underset{,}{\boldsymbol{a}} - \rho^w\, \underset{,}{\boldsymbol{b}} \right) \\
&= -\underset{,,}{\boldsymbol{\kappa}}_w \bullet \underset{,}{\mathbf{grad}}\, p_w - \rho^w\, \underset{,,}{\boldsymbol{\kappa}}_w \bullet \underset{,}{\boldsymbol{a}} + \rho^w\, \underset{,,}{\boldsymbol{\kappa}}_w \bullet \underset{,}{\boldsymbol{b}} \\
&= -\boldsymbol{\kappa}\, \bar{\boldsymbol{G}}\, \boldsymbol{P} - \rho^w\, \boldsymbol{\kappa}\, \boldsymbol{S}\, \boldsymbol{A} + \rho^w\, \boldsymbol{\kappa}\, \boldsymbol{b}
\end{aligned}
$$

Neglecting the gradient of water density (with $\bar{C}_w = \frac{\eta}{K_w}$):

$$
\begin{aligned}
0 &= \bar{C}_w\, \dot{p}_w + \operatorname{div} \underset{,}{\boldsymbol{v}} + \frac{1}{\rho^w} \operatorname{div} \left(\eta\, S_w\, \rho^w\, \underset{,}{\boldsymbol{w}}_w\right) \\
0 &= \bar{C}_w\, \dot{p}_w + \operatorname{div} \underset{,}{\boldsymbol{v}} + \operatorname{div} \left(\eta\, \underset{,}{\boldsymbol{w}}_w\right)
\end{aligned}
$$

$$
\int_\Omega W\, \operatorname{div}\left(\eta\, \underset{,}{\boldsymbol{w}}_w\right) \mathrm{d}\Omega = \int_\Gamma \operatorname{div}\left(W\, \eta\, \underset{,}{\boldsymbol{w}}_w\right) \mathrm{d}\Gamma - \int_\Omega \underset{,}{\mathbf{grad}}\, W \bullet \left(\eta\, \underset{,}{\boldsymbol{w}}_w\right) \mathrm{d}\Omega
$$

Coupled Systems: Porous Media

$$
\begin{aligned}
\int_\Omega W\, \operatorname{div}\left(\eta\, \underset{,}{\boldsymbol{w}}_w\right) \mathrm{d}\Omega &= \int_\Gamma W\, \underbrace{\eta\, \underset{,}{\boldsymbol{w}}_w \bullet \underset{,}{\hat{\boldsymbol{n}}}}_{\bar{q}_w}\, \mathrm{d}\Gamma - \int_\Omega \underset{,}{\mathbf{grad}}\, W \bullet \left(\eta\, \underset{,}{\boldsymbol{w}}_w\right) \mathrm{d}\Omega \\
&= \int_\Gamma W\, \bar{q}_w\, \mathrm{d}\Gamma - \int_\Omega \underset{,}{\mathbf{grad}}\, W \bullet \left(\eta\, \underset{,}{\boldsymbol{w}}_w\right) \mathrm{d}\Omega \\
&= \sum_n W_n \left(\int_\Gamma S_n\, \bar{q}_w\, \mathrm{d}\Gamma - \int_\Omega \underset{,}{\bar{\boldsymbol{G}}}_n \bullet \left(\eta\, \underset{,}{\boldsymbol{w}}_w\right) \mathrm{d}\Omega \right)
\end{aligned}
$$

$$
\begin{aligned}
\int_\Omega \underset{,}{\bar{\boldsymbol{G}}}_n \bullet \left(\eta\, \underset{,}{\boldsymbol{w}}_w\right) \mathrm{d}\Omega &\equiv \int_\Omega \bar{\boldsymbol{G}}^T \left(-\boldsymbol{\kappa}\, \bar{\boldsymbol{G}}\, \boldsymbol{P} - \rho^w\, \boldsymbol{\kappa}\, \boldsymbol{S}\, \boldsymbol{A} + \rho^w\, \boldsymbol{\kappa}\, \boldsymbol{b}\right) \mathrm{d}\Omega \\
&= -\int_\Omega \bar{\boldsymbol{G}}^T \boldsymbol{\kappa}\, \bar{\boldsymbol{G}}\, \boldsymbol{P}\, \mathrm{d}\Omega - \int_\Omega \rho^w\, \bar{\boldsymbol{G}}^T \boldsymbol{\kappa}\, \boldsymbol{S}\, \boldsymbol{A}\, \mathrm{d}\Omega + \int_\Omega \rho^w\, \bar{\boldsymbol{G}}^T \boldsymbol{\kappa}\, \boldsymbol{b}\, \mathrm{d}\Omega
\end{aligned}
$$

Coupled Systems: Porous Media

$$0 = \bar{C}_w \, \dot{p}_w + \dot{\varepsilon}_v + \mathrm{div}\,(\eta \, \underset{\sim}{\boldsymbol{w}}_w)$$

$$\mathrm{div}\, \underset{\sim}{\boldsymbol{v}} = \dot{\varepsilon}_v = \boldsymbol{I}^T \boldsymbol{B}\, \boldsymbol{V} \quad \text{and} \quad \dot{p}_w = \bar{\boldsymbol{S}}\, \dot{\boldsymbol{P}}$$

$$\begin{aligned}
0 = & \int_{\Omega^e} \bar{\boldsymbol{S}}^T \bar{C}_w \dot{p}_w \, \mathrm{d}\Omega \\
& + \int_{\Omega^e} \bar{\boldsymbol{S}}^T \dot{\varepsilon}_v \, \mathrm{d}\Omega \\
& - \int_{\Omega^e} \bar{\boldsymbol{G}}^T (\eta \, \underset{\sim}{\boldsymbol{w}}_w) \, \mathrm{d}\Omega \\
& + \int_{\Omega^e} \bar{\boldsymbol{S}}^T \bar{q}_w \, \mathrm{d}\Omega
\end{aligned}$$

$$\begin{aligned}
0 = & \int_{\Omega^e} \bar{C}_w \bar{\boldsymbol{S}}^T \bar{\boldsymbol{S}}\, \dot{\boldsymbol{P}}^e \, \mathrm{d}\Omega & 0 = & \boldsymbol{L}^e \, \dot{\boldsymbol{P}}^e \\
& + \int_{\Omega^e} \bar{\boldsymbol{S}}^T \boldsymbol{I}^T \boldsymbol{B}\, \boldsymbol{V}^e \, \mathrm{d}\Omega & & + \bar{\boldsymbol{Q}}^e \, \boldsymbol{V}^e \\
& + \int_{\Omega^e} \bar{\boldsymbol{G}}^T \boldsymbol{\kappa}\, \bar{\boldsymbol{G}}\, \boldsymbol{P}^e \, \mathrm{d}\Omega & & + \boldsymbol{H}^e \, \boldsymbol{P}^e \\
& + \int_{\Omega^e} \rho^w \, \bar{\boldsymbol{G}}^T \boldsymbol{\kappa}\, \boldsymbol{S} \boldsymbol{A}^e \, \mathrm{d}\Omega & & + \boldsymbol{O}^e \, \boldsymbol{A}^e \\
& - \int_{\Omega^e} \rho^w \, \bar{\boldsymbol{G}}^T \boldsymbol{\kappa}\, \boldsymbol{b} \, \mathrm{d}\Omega & & - \bar{\boldsymbol{F}}^e \\
& + \int_{\Omega^e} \boldsymbol{S}^T \bar{q}_w \, \mathrm{d}\Omega & &
\end{aligned}$$

Coupled Systems: Porous Media

Thus:

$$\boxed{\boldsymbol{O}^e \, \boldsymbol{A}^e + \bar{\boldsymbol{Q}}^e \, \boldsymbol{V}^e + \boldsymbol{L}^e \, \dot{\boldsymbol{P}}^e + \boldsymbol{H}^e \, \boldsymbol{P}^e = \bar{\boldsymbol{F}}^e}$$

With:

dynamic seepage $\boldsymbol{O}^e = \int_{\Omega^e} \rho^w \, \bar{\boldsymbol{G}}^T \boldsymbol{\kappa}\, \boldsymbol{S} \, \mathrm{d}\Omega$

coupling $\bar{\boldsymbol{Q}}^e = \int_{\Omega^e} \bar{\boldsymbol{S}}^T \boldsymbol{I}^T \boldsymbol{B} \, \mathrm{d}\Omega$

compressibility $\boldsymbol{L}^e = \int_{\Omega^e} \bar{C}_w \bar{\boldsymbol{S}}^T \bar{\boldsymbol{S}} \, \mathrm{d}\Omega$

conductivity $\boldsymbol{H}^e = \int_{\Omega^e} \bar{\boldsymbol{G}}^T \boldsymbol{\kappa}\, \bar{\boldsymbol{G}} \, \mathrm{d}\Omega$

flow vector $\bar{\boldsymbol{F}}^e = \int_{\Omega^e} \rho^w \, \bar{\boldsymbol{G}}^T \boldsymbol{\kappa}\, \boldsymbol{b} \, \mathrm{d}\Omega - \int_{\Omega^e} \boldsymbol{S}^T \bar{q}_w \, \mathrm{d}\Omega$

And:

$$\bar{\boldsymbol{Q}}^e = (\boldsymbol{Q}^e)^T$$

Porous Media: time solution

Fundamentals

Coupled Systems
Porous media 1
Porous media 2
Porous media 3
Porous media 4
Porous media 5
Porous media 6
Porous media 7
Porous media 8
Porous media 9
Porous media 10
▷ Time solution 1
Time solution 2
Time solution 3
Time solution 4
Time solution 5

After assemblage:

$$\boldsymbol{MA} + \boldsymbol{CV} + \boldsymbol{KU} - \boldsymbol{QP} = \boldsymbol{F}$$
$$\boldsymbol{OA} + \bar{\boldsymbol{Q}}\boldsymbol{V} + \boldsymbol{L}\dot{\boldsymbol{P}} + \boldsymbol{HP} = \bar{\boldsymbol{F}}$$

Introducing the boundary conditions:

$$\begin{bmatrix} \boldsymbol{M}_{11} & \boldsymbol{M}_{12} \\ \boldsymbol{M}_{21} & \boldsymbol{M}_{22} \end{bmatrix} \begin{Bmatrix} \dot{\boldsymbol{V}}_1 \\ \dot{\boldsymbol{V}}_2 \end{Bmatrix} + \begin{bmatrix} \boldsymbol{C}_{11} & \boldsymbol{C}_{12} \\ \boldsymbol{C}_{21} & \boldsymbol{C}_{22} \end{bmatrix} \begin{Bmatrix} \boldsymbol{V}_1 \\ \boldsymbol{V}_2 \end{Bmatrix} + \begin{bmatrix} \boldsymbol{K}_{11} & \boldsymbol{K}_{12} \\ \boldsymbol{K}_{21} & \boldsymbol{K}_{22} \end{bmatrix} \begin{Bmatrix} \boldsymbol{U}_1 \\ \boldsymbol{U}_2 \end{Bmatrix} - \begin{bmatrix} \boldsymbol{Q}_{1a} & \boldsymbol{Q}_{1b} \\ \boldsymbol{Q}_{2a} & \boldsymbol{Q}_{2b} \end{bmatrix} \begin{Bmatrix} \boldsymbol{P}_a \\ \boldsymbol{P}_b \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{F}_1 \\ \boldsymbol{F}_2 \end{Bmatrix}$$

$$\underbrace{\boldsymbol{M}_{11}}_{\boldsymbol{M}} \underbrace{\dot{\boldsymbol{V}}_1}_{\dot{\boldsymbol{v}}} + \underbrace{\boldsymbol{C}_{11}}_{\boldsymbol{C}} \underbrace{\boldsymbol{V}_1}_{\boldsymbol{v}} + \underbrace{\boldsymbol{K}_{11}}_{\boldsymbol{K}} \underbrace{\boldsymbol{U}_1}_{\boldsymbol{u}} - \underbrace{\boldsymbol{Q}_{1a}}_{\boldsymbol{Q}} \underbrace{\boldsymbol{P}_a}_{\boldsymbol{p}} = \underbrace{\boldsymbol{F}_1 - \boldsymbol{M}_{12}\dot{\boldsymbol{V}}_2 - \boldsymbol{C}_{12}\boldsymbol{V}_2 - \boldsymbol{K}_{12}\boldsymbol{U}_2 + \boldsymbol{Q}_{1b}\boldsymbol{P}_b}_{\boldsymbol{g}}$$

Or:

$$\boldsymbol{M}\dot{\boldsymbol{v}} + \boldsymbol{C}\boldsymbol{v} + \boldsymbol{K}\boldsymbol{u} - \boldsymbol{Q}\boldsymbol{p} = \boldsymbol{g}$$

Porous Media: time solution

Fundamentals

Coupled Systems
Porous media 1
Porous media 2
Porous media 3
Porous media 4
Porous media 5
Porous media 6
Porous media 7
Porous media 8
Porous media 9
Porous media 10
Time solution 1
▷ Time solution 2
Time solution 3
Time solution 4
Time solution 5

Introducing boundary conditions to the other system:

$$\begin{bmatrix} \boldsymbol{O}_{a1} & \boldsymbol{O}_{a2} \\ \boldsymbol{O}_{b1} & \boldsymbol{O}_{b2} \end{bmatrix} \begin{Bmatrix} \dot{\boldsymbol{V}}_1 \\ \dot{\boldsymbol{V}}_2 \end{Bmatrix} + \begin{bmatrix} \bar{\boldsymbol{Q}}_{a1} & \bar{\boldsymbol{Q}}_{a2} \\ \bar{\boldsymbol{Q}}_{b1} & \bar{\boldsymbol{Q}}_{b2} \end{bmatrix} \begin{Bmatrix} \boldsymbol{V}_1 \\ \boldsymbol{V}_2 \end{Bmatrix} + \begin{bmatrix} \boldsymbol{L}_{aa} & \boldsymbol{L}_{ab} \\ \boldsymbol{L}_{ba} & \boldsymbol{L}_{bb} \end{bmatrix} \begin{Bmatrix} \dot{\boldsymbol{P}}_a \\ \dot{\boldsymbol{P}}_b \end{Bmatrix} + \begin{bmatrix} \boldsymbol{H}_{aa} & \boldsymbol{H}_{ab} \\ \boldsymbol{H}_{ba} & \boldsymbol{H}_{bb} \end{bmatrix} \begin{Bmatrix} \boldsymbol{P}_a \\ \boldsymbol{P}_b \end{Bmatrix} = \begin{Bmatrix} \bar{\boldsymbol{F}}_a \\ \bar{\boldsymbol{F}}_b \end{Bmatrix}$$

$$\underbrace{\boldsymbol{O}_{a1}}_{\boldsymbol{O}} \underbrace{\dot{\boldsymbol{V}}_1}_{\dot{\boldsymbol{v}}} + \underbrace{\bar{\boldsymbol{Q}}_{a1}}_{\bar{\boldsymbol{Q}}} \underbrace{\boldsymbol{V}_1}_{\boldsymbol{v}} + \underbrace{\boldsymbol{L}_{aa}}_{\boldsymbol{L}} \underbrace{\dot{\boldsymbol{P}}_a}_{\dot{\boldsymbol{p}}} + \underbrace{\boldsymbol{H}_{aa}}_{\boldsymbol{H}} \underbrace{\boldsymbol{P}_a}_{\boldsymbol{p}} = \underbrace{\bar{\boldsymbol{F}}_a - \boldsymbol{O}_{a2}\dot{\boldsymbol{V}}_2 - \bar{\boldsymbol{Q}}_{a2}\boldsymbol{V}_2 - \boldsymbol{L}_{ab}\dot{\boldsymbol{P}}_b - \boldsymbol{H}_{ab}\boldsymbol{P}_b}_{\bar{\boldsymbol{g}}}$$

Or:

$$\boldsymbol{O}\dot{\boldsymbol{v}} + \bar{\boldsymbol{Q}}\boldsymbol{v} + \boldsymbol{L}\dot{\boldsymbol{p}} + \boldsymbol{H}\boldsymbol{p} = \bar{\boldsymbol{g}}$$

For an updated time, from t_0 to t_1:

$$\boldsymbol{M}\dot{\boldsymbol{v}}_1 + \boldsymbol{C}\boldsymbol{v}_1 + \boldsymbol{K}\boldsymbol{u}_1 - \boldsymbol{Q}\boldsymbol{p}_1 = \boldsymbol{g}_1$$
$$\boldsymbol{O}\dot{\boldsymbol{v}}_1 + \bar{\boldsymbol{Q}}\boldsymbol{v}_1 + \boldsymbol{L}\dot{\boldsymbol{p}}_1 + \boldsymbol{H}\boldsymbol{p}_1 = \bar{\boldsymbol{g}}_1$$

The θ-method for the water pressure is:

$$\boldsymbol{p}_1 = \boldsymbol{p}_0 + (1-\theta)\, h\, \dot{\boldsymbol{p}}_0 + \theta\, h\, \dot{\boldsymbol{p}}_1$$

hence:

$$\dot{\boldsymbol{p}}_1 = \underbrace{\frac{1}{\theta\, h}}_{\beta_1} \boldsymbol{p}_1 - \underbrace{\frac{1}{\theta\, h}}_{\beta_1} \boldsymbol{p}_0 - \underbrace{\frac{1-\theta}{\theta}}_{\beta_2} \dot{\boldsymbol{p}}_0$$

or:

$$\boxed{\dot{\boldsymbol{p}}_1 = \beta_1\, \boldsymbol{p}_1 - \boldsymbol{o}}$$

with:

$$\boxed{\boldsymbol{o} = \beta_1\, \boldsymbol{p}_0 + \beta_2\, \dot{\boldsymbol{p}}_0}$$

the other definitions for the Newmark's method are:

$$\begin{aligned} \boldsymbol{m} &= \alpha_1\, \boldsymbol{u}_0 + \alpha_2\, \boldsymbol{v}_0 + \alpha_3\, \dot{\boldsymbol{v}}_0 \\ \boldsymbol{n} &= \alpha_4\, \boldsymbol{u}_0 + \alpha_5\, \boldsymbol{v}_0 + \alpha_6\, \dot{\boldsymbol{v}}_0 \\ \dot{\boldsymbol{v}}_1 &= \alpha_1\, \boldsymbol{u}_1 - \boldsymbol{m} \\ \boldsymbol{v}_1 &= \alpha_4\, \boldsymbol{u}_1 - \boldsymbol{n} \end{aligned}$$

Employing Newmark's method for the second order equation:

$$\boldsymbol{M} \cdot (\alpha_1\, \boldsymbol{u}_1 - \boldsymbol{m}) + \boldsymbol{C} \cdot (\alpha_4\, \boldsymbol{u}_1 - \boldsymbol{n}) + \boldsymbol{K}\boldsymbol{u}_1 - \boldsymbol{Q}\boldsymbol{p}_1 = \boldsymbol{g}_1$$

and the θ-method for the first order equation:

$$\boldsymbol{O} \cdot (\alpha_1\, \boldsymbol{u}_1 - \boldsymbol{m}) + \bar{\boldsymbol{Q}} \cdot (\alpha_4\, \boldsymbol{u}_1 - \boldsymbol{n}) + \boldsymbol{L} \cdot (\beta_1\, \boldsymbol{p}_1 - \boldsymbol{o}) + \boldsymbol{H}\boldsymbol{p}_1 = \bar{\boldsymbol{g}}_1$$

Thus:

$$(\alpha_1 \boldsymbol{M} + \alpha_4 \boldsymbol{C} + \boldsymbol{K})\, \boldsymbol{u}_1 - \boldsymbol{Q}\boldsymbol{p}_1 = \boldsymbol{g}_1 + \boldsymbol{M}\boldsymbol{m} + \boldsymbol{C}\boldsymbol{n}$$

and:

$$(\alpha_1 \boldsymbol{O} + \alpha_4 \bar{\boldsymbol{Q}})\, \boldsymbol{u}_1 + (\beta_1 \boldsymbol{L} + \boldsymbol{H})\, \boldsymbol{p}_1 = \bar{\boldsymbol{g}}_1 + \boldsymbol{O}\boldsymbol{m} + \bar{\boldsymbol{Q}}\boldsymbol{n} + \boldsymbol{L}\boldsymbol{o}$$

Or, in matrix form:

$$\begin{bmatrix} \alpha_1 \boldsymbol{M} + \alpha_4 \boldsymbol{C} + \boldsymbol{K} & -\boldsymbol{Q} \\ \alpha_1 \boldsymbol{O} + \alpha_4 \bar{\boldsymbol{Q}} & \beta_1 \boldsymbol{L} + \boldsymbol{H} \end{bmatrix} \begin{Bmatrix} \boldsymbol{u}_1 \\ \boldsymbol{p}_1 \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{g}_1 + \boldsymbol{M}\boldsymbol{m} + \boldsymbol{C}\boldsymbol{n} \\ \bar{\boldsymbol{g}}_1 + \boldsymbol{O}\boldsymbol{m} + \bar{\boldsymbol{Q}}\boldsymbol{n} + \boldsymbol{L}\boldsymbol{o} \end{Bmatrix}$$

If the dynamic component of the second equation is neglected (dropping $\boldsymbol{O}$) and dividing the second equation by $-\alpha_4$, the system becomes symmetric:

$$\begin{bmatrix} \alpha_1 \boldsymbol{M} + \alpha_4 \boldsymbol{C} + \boldsymbol{K} & -\boldsymbol{Q} \\ -\bar{\boldsymbol{Q}} & \frac{-\beta_1}{\alpha_4} \boldsymbol{L} - \frac{1}{\alpha_4} \boldsymbol{H} \end{bmatrix} \begin{Bmatrix} \boldsymbol{u}_1 \\ \boldsymbol{p}_1 \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{g}_1 + \boldsymbol{M}\boldsymbol{m} + \boldsymbol{C}\boldsymbol{n} \\ \frac{-1}{\alpha_4} (\bar{\boldsymbol{g}}_1 + \bar{\boldsymbol{Q}}\boldsymbol{n} + \boldsymbol{L}\boldsymbol{o}) \end{Bmatrix}$$