In [3]:
```
%autosave 0
```

```
Autosave disabled
```

In [1]:
```
!pip3 install scikit-fem
```

```
Collecting scikit-fem
  Downloading scikit_fem-8.0.0-py3-none-any.whl (157 kB)
     ---------------------------------------- 157.8/157.8 kB 3.1 MB/s eta 0:00:00
Requirement already satisfied: scipy in c:\users\gary\appdata\local\programs\pytho
n\python39\lib\site-packages (from scikit-fem) (1.9.3)
Requirement already satisfied: numpy in c:\users\gary\appdata\local\programs\pytho
n\python39\lib\site-packages (from scikit-fem) (1.23.5)
Installing collected packages: scikit-fem
Successfully installed scikit-fem-8.0.0
```

Chapter 31-15 Solving PDEs with the Finite Element Method.

The finite element method (FEM) is a popular method for numerically solving differential equations arising in engineering and mathematical modeling. Typical problem areas of interest include the traditional fields of structural analysis, heat transfer, fluid flow, mass transport, and electromagnetic potential.

The FEM is a general numerical method for solving partial differential equations in two or three space variables (i.e., some boundary value problems). To solve a problem, the FEM subdivides a large system into smaller, simpler parts that are called finite elements. This is achieved by a particular space discretization in the space dimensions, which is implemented by the construction of a mesh of the object: the numerical domain for the solution, which has a finite number of points. The finite element method formulation of a boundary value problem finally results in a system of algebraic equations. The method approximates the unknown function over the domain. The simple equations that model these finite elements are then assembled into a larger system of equations that models the entire problem. The FEM then approximates a solution by minimizing an associated error function via the calculus of variations.

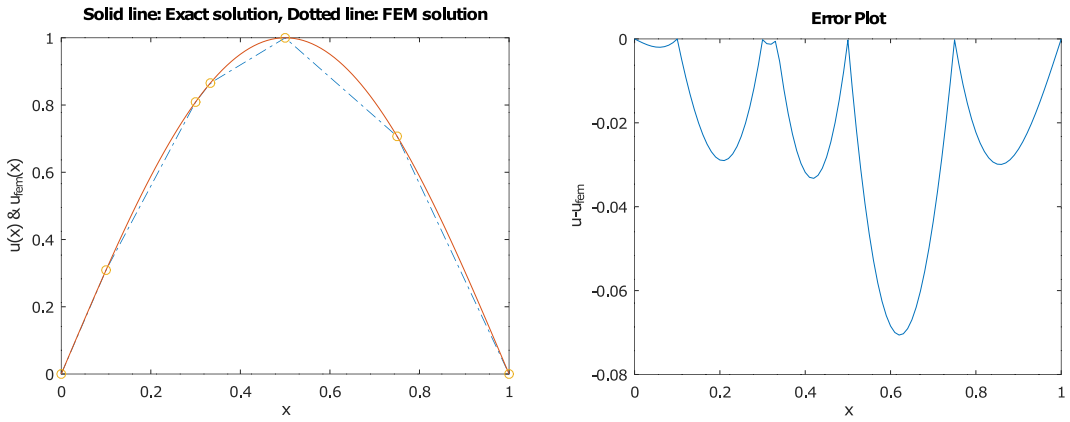
from Wikipedia

1. This problem depends on material from Chapters 6 and 8 of the book *Numerical Solution of Differential Equations -- Introduction to Finite Difference and Finite Element Methods* by Li, Qiao, and Tang. At the address *https://zhilin.math.ncsu.edu/* is Li's page featuring details about the book. There is a button on the page, just underneath the book title, with the label "Matlab Codes". The download will be entitled "DE_FEM_Matlab_Code.zip" All of the files in the folder with the title "Chapter 6" should be loaded into Octave. Only one will execute in Octave, the one entitled **drive.m**. This file is shown below.

In [ ]:
```
clear all; close all;   % Clear every thing so it won't mess up with other
                        % existing variables.

%%%%%% Generate a triangulation

 x(1)=0; x(2)=0.1; x(3)=0.3; x(4)=0.333; x(5)=0.5; x(6)=0.75;x(7)=1;

%x=0:0.05:1;

%%%%%% Call fem1d function to get the FEM solution at nodal points.

U = fem1d(x);

%%%%%% Compare errors:

x2 = 0:0.01:1; k2 = length(x2);
for i=1:k2,
  u_exact(i) = soln(x2(i));
  u_fem(i) = fem_soln(x,U,x2(i));  % Compute FEM solution at x2(i)
end

error = norm(u_fem-u_exact,inf)  % Compute the infinity error

plot(x2,u_fem,'-.', x2,u_exact)   % Solid: the exact, %dotted: FEM solution
hold; plot(x,U,'o')              % Mark the solution at nodal points.
% set(gca,'FontSize',18);
xlabel('x'); ylabel('u(x) & u_{fem}(x)');
title('Solid line: Exact solution, Dotted line: FEM solution')

figure(2); % set(gca,'FontSize',18);
plot(x2,u_fem-u_exact); title('Error plot')
xlabel('x'); ylabel('u-u_{fem}');  title('Error Plot')
```

If access to the book mentioned above is possible, some good info regarding the relation between PDEs and finite element modeling can be learned.

The plots which are produced by running the code in the above cell are shown below.

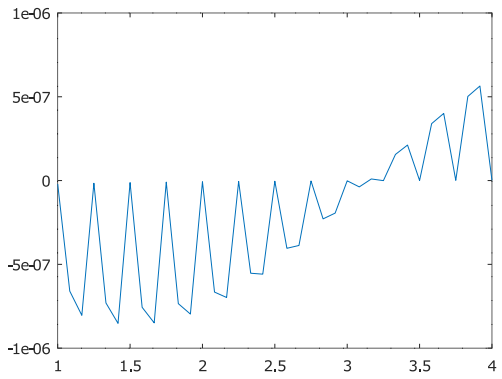


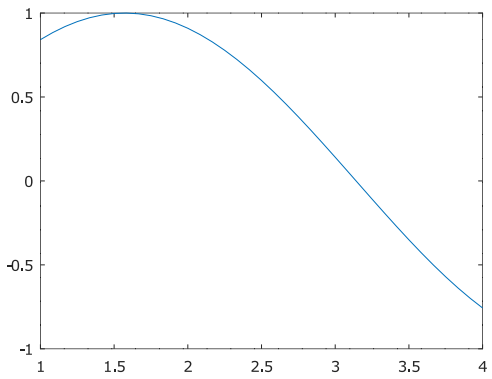
3. For the initial conditions shown in the comment banner in the cell below, construct a 1-dimensional finite element solution and plot.

The plot below on the left shows the error; the plot on the right shows the analytical solution.

In [ ]:
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                              %
%  Matlab program for one-dimensinal finite element method for                 %
%                                                                              %
%    -(k(x)u_x)_x + c(x) u_x + b(x) u(x) = f(x)                                %
%                                                                              %
%  with different boundary condition at x =0, and x = l                        %
%                                                                              %
%------------------------------------------------------------------------------%

        function [x,u]= fem1d

   clear
%  Preprocessor:

   global nnode nelem
   global gk gf
   global xi  w

%%%%%%%%% Start the program %%%%%%%%%
```

```
%   clear

   [xi,w] = setint;   % Get Gaussian points and weights.

%  Preprocessor:
   [x,kbc,vbc,kind,nint,nodes] = propset;             % Input data

    formkf(kind,nint,nodes,x,xi,w);   % Form the discrete system

    aplybc(kbc,vbc);

    u = gk\gf;                   % Solve the linear system.

    for i=1:nnode,
      e(i) = u(i) - uexact(x(i));
    end
    figure(1); plot(x,e)
    figure(2); plot(x,u)
```





In [ ]:

Below is shown a gallery of residual error results. It was assembled in Inkscape. How to assemble a simlar composite image. The font size in Octave may look tiny, but when it reaches Inkscape it may be huge. In this particular case it was advisable to zoom in to an overall size of about 4×6 before saving the file. Then on opening in Inkscape the fonts look normal. The object is reduced to a consistent numerical zoom, in this case 45%. Then the object is copied to clipboard, and the object itself is deleted. The next saved Octave plot is opened in Inkscape, and reduced to exactly the same zoom level as was used for the first. Then the first can be pasted back into the scene. The two

objects are positioned as required; selecting them individually will show a dotted border, and the two dotted borders should be colinear at their adjacent edges. With both objects selected, the command 'Group' from the Object menu freezes them together temporarily. But to make them into a single indivisible object the command 'Object to Path', followed by 'Union', (both commands found in the Path menu), must be given. After that the two objects are permanently joined. Additional objects are opened, scaled, copied, destroyed, pasted, aligned, grouped, converted to paths, and unioned, until the composite object looks as desired. It is also necessary to size the background document outline until the composite object justs fits with thin border, for centered image, or with large right or left border for positioning appropriately on the page. If the size of the object is (unintuitively) reduced, it will appear larger in relation to the background document outline, and will look larger when finally pasted into Jupyter. If the working image is saved into .SVG in Inkscape and then pasted into Jupyter, it will become an integral part of the notebook, and the notebook will stand alone, requiring no accessory files.

In [ ]:
```
function Poisson_P2
%% OK!
% -nu*\laplace u=f in \Omega
%  u=0 on \partial\Omega
% u=x(1-x)y(1-y)
% f=2*nu*(y(1-y)+x(1-x))

for level=0:4
    n=1*2^level;
    h=1/n;

    N=2*n;
    H=1/N;
    nele=2*n^2;
    nunk=(N+1)^2;

    for node=1:nunk %% coordinates of P2
        i=mod(node-1,N+1);  % line
        j=fix((node-1)/(N+1));   % collu
        coordinate(node,1)=i*H;
        coordinate(node,2)=j*H;
    end

    element(1,1) = 1;
    element(1,2) = 3;
```

```
    element(1,3) = 2*N + 3;
    element(1,4) = 2;
    element(1,5) = N + 3;
    element(1,6) = N + 2;

    element(2,1) = 2*N + 5;
    element(2,2) = 2*N + 3;
    element(2,3) = 3;
    element(2,4) = 2*N + 4;
    element(2,5) = N + 3;
    element(2,6) = N + 4;
    for i = 3 : 2 * n
        element(i,1:6) = element(i-2,1:6) + 2;
    end
    for i = 2 * n + 1 : nele
        element(i,1:6) = element(i-2*n,1:6) + 2 * ( N + 1);
    end
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    a=sparse(nunk,nunk);
    f=sparse(nunk,1);
    [a, f] = assemble( element, nele, nunk, n );

    u=a\f;

    uu=exact(coordinate, nunk);
    norm(uu-(u(1:nunk))')

    %% draw figures %%
    x=0:H:1;
    y=0:H:1;
    figure(n);
    perr=reshape(abs(uu-u'),N+1,N+1);
    surf(y,x,perr);
    figure(n+1);
    up=reshape(u',N+1,N+1);
    surf(y,x,up);
end

save Poisson.mat
```

The Python module scikit-fem is a lightweight but sturdy implement for finite element calculations in Python. It doesn't need compiling, and also entails no dependencies for the average Python installation. This is in contrast to others like Fenics, NGSolve, and their ilk. The ten cells below contain the first exercise in the Getting Started section of the scikit-fem documentation.

4. Solve the Poisson problem

$$-\Delta u = f \qquad \text{in } \Omega$$

$$u = 0 \qquad \text{on } \partial\Omega$$

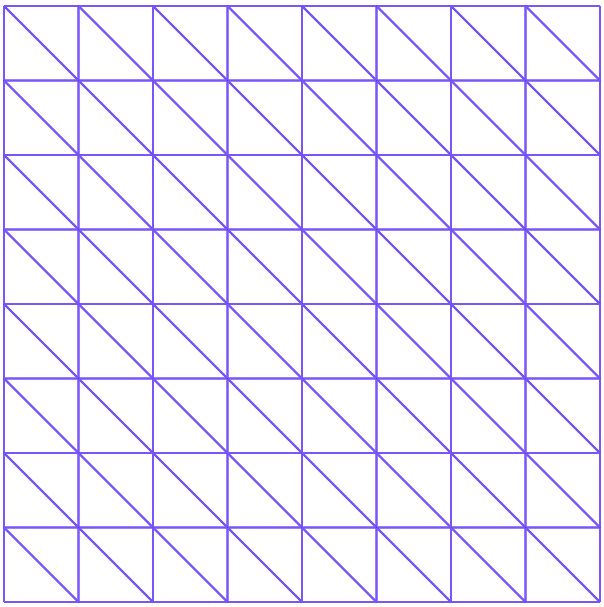
where $\Omega = (0,1)^2$ is a square domain and $f(x,y) = \sin \pi x \sin \pi y$

In [1]:
```
import skfem as fem
>>> from skfem.helpers import dot, grad  # helpers make forms look nice
>>> @fem.BilinearForm
... def a(u, v, _):
...     return dot(grad(u), grad(v))
```

In [2]:
```
import numpy as np
>>> @fem.LinearForm
... def L(v, w):
...     x, y = w.x  # global coordinates
...     f = np.sin(np.pi * x) * np.sin(np.pi * y)
...     return f * v
```

In [4]:
```
mesh = fem.MeshTri().refined(3)  # refine thrice
mesh
```

Out[4]:



In [6]:
```
Vh = fem.Basis(mesh, fem.ElementTriP1())
Vh
```

Out[6]:
```
<skfem CellBasis(MeshTri1, ElementTriP1) object>
  Number of elements: 128
  Number of DOFs: 81
  Size: 27648 B
```

In [9]:
```
A = a.assemble(Vh)
l = L.assemble(Vh)
A.shape
```

Out[9]: (81, 81)

In [10]:
```
l.shape
```

Out[10]: (81,)

In [11]:
```
D = Vh.get_dofs()
D
```

Out[11]:
```
<skfem DofsView(MeshTri1, ElementTriP1) object>
  Number of nodal DOFs: 32 ['u']
```

In [13]:
```
x = fem.solve(*fem.condense(A, l, D=D))
x.shape
```

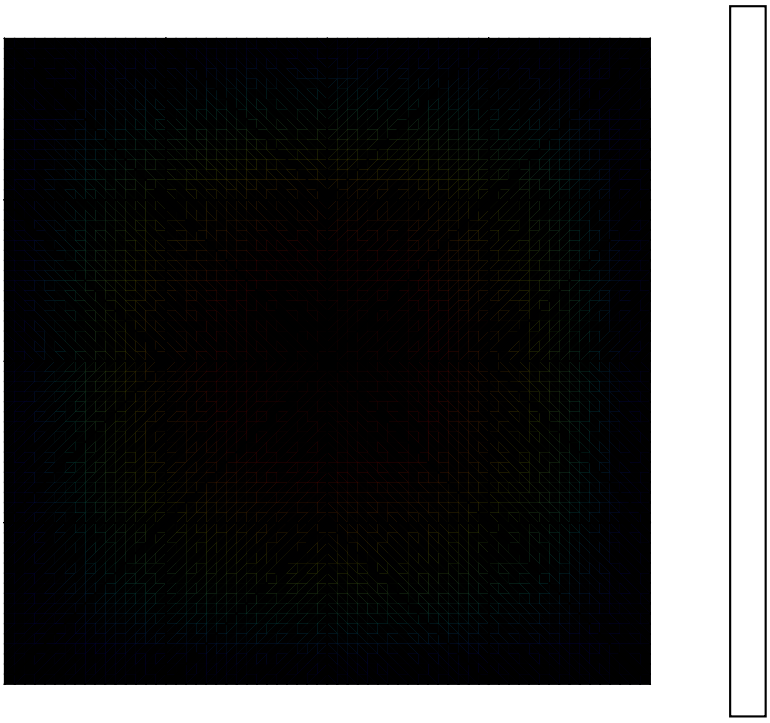
Out[13]: (81,)

In [15]:
```
@fem.Functional
def error(w):
    x, y = w.x
    uh = w['uh']
    u = np.sin(np.pi * x) * np.sin(np.pi * y) / (2. * np.pi ** 2)
    return (uh - u) ** 2
round(error.assemble(Vh, uh=Vh.interpolate(x)), 9)
```

Out[15]: 1.069e-06

In [17]:
```
"""Discontinuous Galerkin method."""

from skfem import *
from skfem.helpers import grad, dot, jump
from skfem.models.poisson import laplace, unit_load

m = MeshTri.init_sqsymmetric().refined()
e = ElementTriDG(ElementTriP4())
alpha = 1e-3

ib = Basis(m, e)
bb = FacetBasis(m, e)
fb = [InteriorFacetBasis(m, e, side=i) for i in [0, 1]]

@BilinearForm
def dgform(u, v, p):
    ju, jv = jump(p, u, v)
    h = p.h
    n = p.n
    return ju * jv / (alpha * h) - dot(grad(u), n) * jv - dot(grad(v), n) * ju

@BilinearForm
def nitscheform(u, v, p):
    h = p.h
    n = p.n
    return u * v / (alpha * h) - dot(grad(u), n) * v - dot(grad(v), n) * u
```

```
A = asm(laplace, ib)
B = asm(dgform, fb, fb)
C = asm(nitscheform, bb)
b = asm(unit_load, ib)

x = solve(A + B + C, b)

M, X = ib.refinterp(x, 4)

def visualize():
    from skfem.visuals.matplotlib import plot, draw
    %config InlineBackend.figure_formats = ['svg']

    ax = draw(M, boundaries_only=True)
    return plot(M, X, shading="gouraud", ax=ax, colorbar=True)

if __name__ == "__main__":
    visualize().show()
```



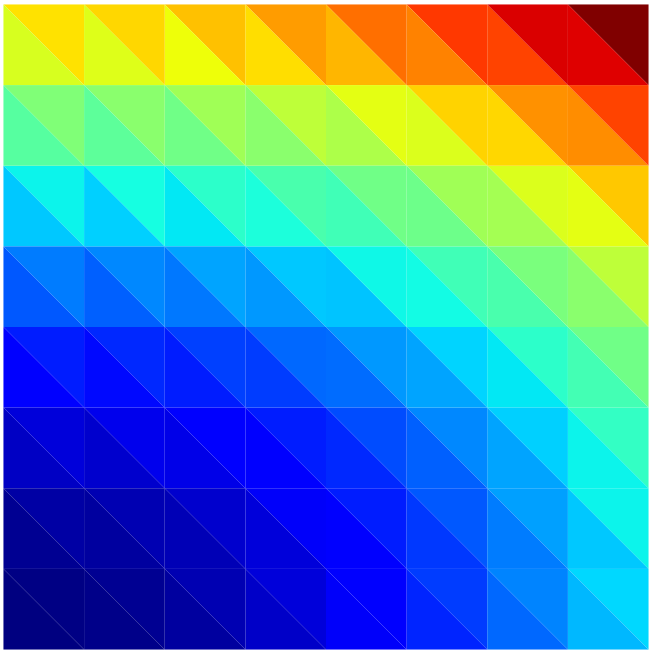
The Github repository of gdmcbain contains eight exercises from the Fenics tutorial book. Since Fenics is such a popular package, it is natural for someone to wonder if the new and somewhat brash program scikit-fem could reproduce the results of the Fenics tutorials. Below is the first of the collection.

Fenics #1 below

In [8]:

```
import numpy as np

from skfem import *
from skfem.models.poisson import laplace, unit_load, mass

#mesh = fem.MeshTri().refined(3)
mesh = MeshTri().refined(3)
mesh

V = InteriorBasis(mesh, ElementTriP1())

u_D = 1 + [1, 2] @ mesh.p ** 2

boundary = mesh.boundary_nodes()

u = np.zeros_like(u_D)
u[boundary] = u_D[boundary]

a = asm(laplace, V)
L = -6.0 * asm(unit_load, V)

u = solve(*condense(a, L, u, D=boundary))

ax = mesh.plot(u)
ax.get_figure().savefig("poisson.svg")

mesh.save("fenics01.ply")

error = u - u_D
print("error_L2  =", np.sqrt(error.T @ asm(mass, V) @ error))
print("error_max =", np.linalg.norm(error, np.inf))
```

```
Warning: PLY doesn't support 64-bit integers. Casting down to 32-bit.
error_L2  = 3.090730095650652e-16
error_max = 1.1102230246251565e-15
```



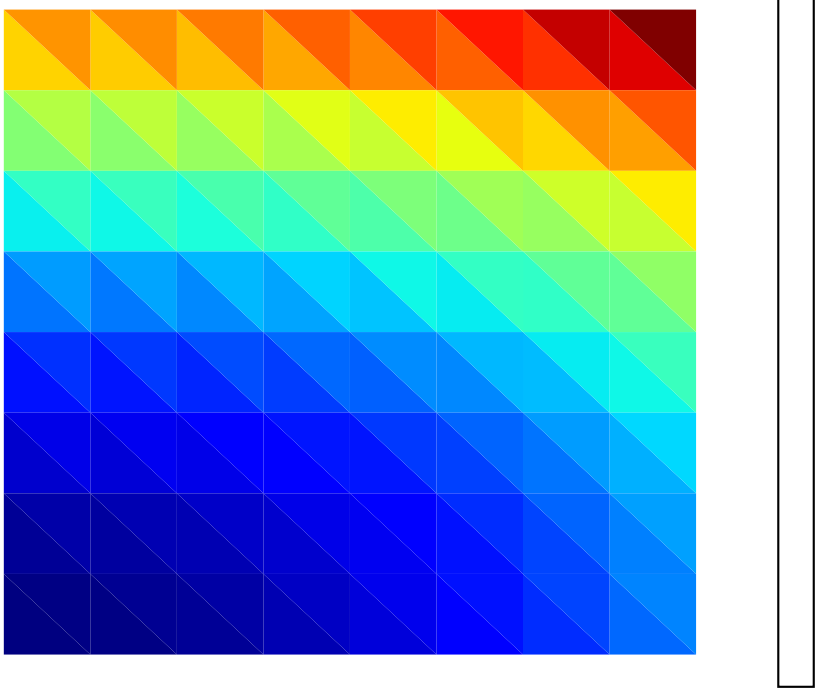
Fenics example #3 below.

In [2]:

```
from matplotlib.pyplot import subplots, pause
import numpy as np
%config InlineBackend.figure_formats = ['svg']

from skfem import *
from skfem.models.poisson import laplace, mass, unit_load

alpha = 3.0
beta = 1.2

nx = ny = 2 ** 3

time_end = 2.0
num_steps = 10
dt = time_end / num_steps

mesh = (
    MeshLine(np.linspace(0, 1, nx + 1)) * MeshLine(np.linspace(0, 1, ny + 1))
).to_meshtri()
basis = InteriorBasis(mesh, ElementTriP1())

boundary = basis.get_dofs().all()
interior = basis.complement_dofs(boundary)

M = asm(mass, basis)
A = M + dt * asm(laplace, basis)
f = (beta - 2 - 2 * alpha) * asm(unit_load, basis)

fig, ax = subplots()
```

```
def dirichlet(t: float) -> np.ndarray:
    return 1.0 + [1.0, alpha] @ mesh.p ** 2 + beta * t


t = 0.0
u = dirichlet(t)

zlim = (0, np.ceil(1 + alpha + beta * time_end))

for i in range(num_steps + 1):

    ax.cla()
    ax.axis("off")
    fig.suptitle("t = {:.4f}".format(t))
    mesh.plot(u, ax=ax, zlim=zlim)
    if t == 0.0:
        fig.colorbar(ax.get_children()[0])
    fig.show()
    pause(1.0)

    t += dt
    b = dt * f + M @ u

    u_D = dirichlet(t)
    u = solve(*condense(A, b, u_D, D=boundary))
    error = np.linalg.norm(u - u_D)
    print("t = %.2f: error = %.3g" % (t, error))
```

```
C:\Users\gary\AppData\Local\Temp/ipykernel_5928/2733724057.py:49: UserWarning: Mat
plotlib is currently using module://matplotlib_inline.backend_inline, which is a n
on-GUI backend, so cannot show the figure.
  fig.show()
```



```
t = 0.20: error = 9.44e-15
t = 0.40: error = 1.25e-14
t = 0.60: error = 1.45e-14
t = 0.80: error = 1.7e-14
t = 1.00: error = 1.68e-14
t = 1.20: error = 1.89e-14
t = 1.40: error = 2.01e-14
t = 1.60: error = 2.01e-14
t = 1.80: error = 2.16e-14
t = 2.00: error = 2.34e-14
t = 2.20: error = 2.46e-14
```

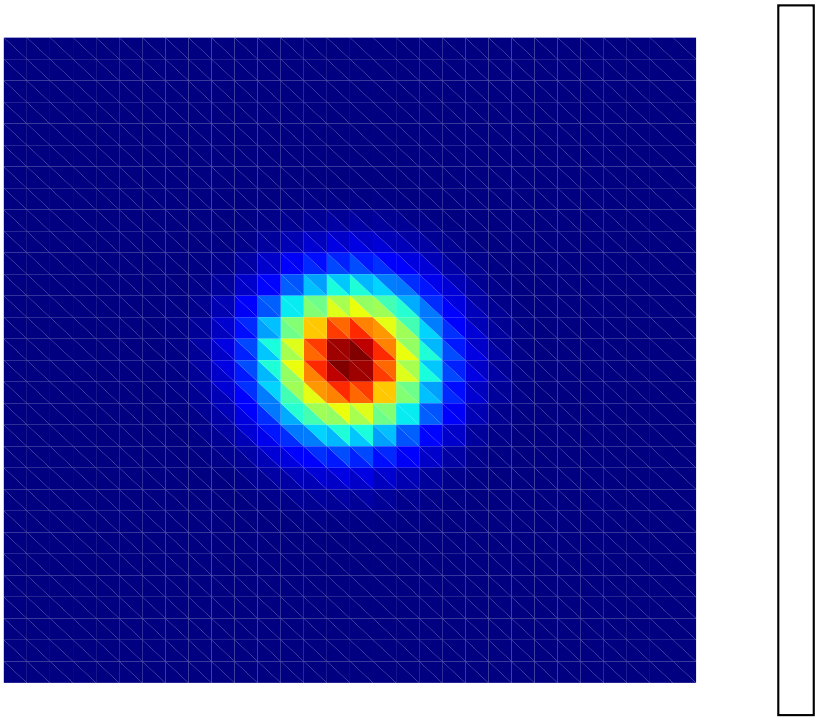
Fenics example #4 below.

In [14]:

```
from pathlib import Path

from matplotlib.pyplot import subplots, pause
import numpy as np

from skfem import *
from skfem.models.poisson import laplace, mass

a = 5.0

nx = ny = 30

time_end = 2.0
num_steps = 50
dt = time_end / num_steps

mesh = (
```

```
    MeshLine(np.linspace(-2, 2, nx + 1)) * MeshLine(np.linspace(-2, 2, ny + 1))
).to_meshtri()
basis = InteriorBasis(mesh, ElementTriP1())

boundary = basis.get_dofs().all()
interior = basis.complement_dofs(boundary)

M = asm(mass, basis)
A = M + dt * asm(laplace, basis)

fig, ax = subplots()

t = 0.0
u = np.exp(-a * (np.sum(mesh.p ** 2, axis=0)))  # initial condition, P1 only

output_dir = Path("heat_gaussian")
try:
    output_dir.mkdir()
except FileExistsError:
    pass

for i in range(num_steps + 1):

    ax.cla()
    ax.axis("off")
    fig.suptitle("t = {:.4f}".format(t))
    mesh.plot(u, ax=ax, zlim=(0, 1))
    if t == 0.0:
        fig.colorbar(ax.get_children()[0])
        fig.savefig("initial.png")
    fig.show()
    pause(0.01)

    t += dt
    b = M @ u

    u = solve(*condense(A, b, D=boundary))

    mesh.save(str(output_dir.joinpath(f"solution{i:06d}.msh")), {"temperature": u})
```

```
C:\Users\gary\AppData\Local\Temp/ipykernel_2196/3973729471.py:48: UserWarning: Mat
plotlib is currently using module://matplotlib_inline.backend_inline, which is a n
on-GUI backend, so cannot show the figure.
  fig.show()
```



Fenics sample #5 below.

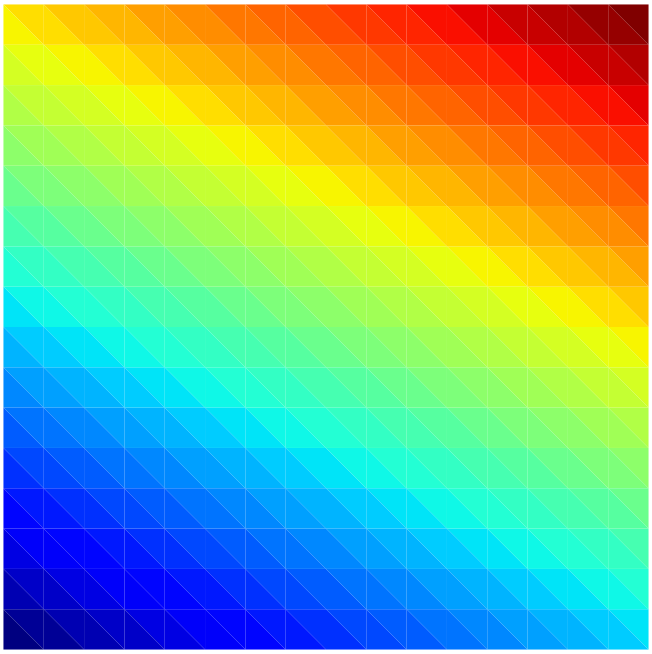
In [18]:

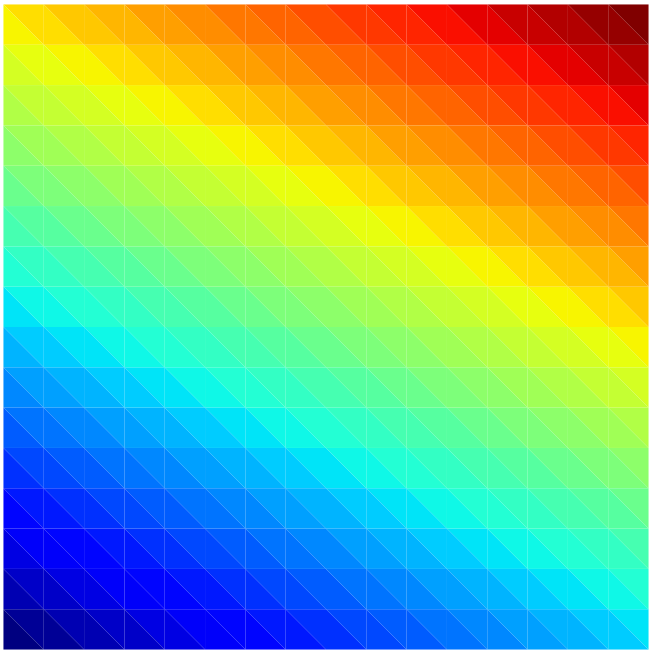
```
from pathlib import Path

import numpy as np
from scipy.optimize import root

from sympy import symbols
from sympy.vector import CoordSys3D, gradient, divergence
from sympy.utilities.lambdify import lambdify

from skfem import (
    MeshTri,
    InteriorBasis,
    ElementTriP1,
    BilinearForm,
    LinearForm,
    asm,
    solve,
    condense,
)
from skfem.models.poisson import laplace
from skfem.visuals.matplotlib import plot
%config InlineBackend.figure_formats = ['svg']


output_dir = Path("poisson_nonlinear")

try:
```

```
    output_dir.mkdir()
except FileExistsError:
    pass


def q(u):
    """Return nonlinear coefficient"""
    return 1 + u * u


R = CoordSys3D("R")


def apply(f, coords):
    x, y = symbols("x y")
    return lambdify((x, y), f.subs({R.x: x, R.y: y}))(*coords)


u_exact = 1 + R.x + 2 * R.y  # exact solution
f = -divergence(q(u_exact) * gradient(u_exact))  # manufactured RHS

mesh = MeshTri().refined(3)  # refine thrice

V = InteriorBasis(mesh, ElementTriP1())

boundary = V.get_dofs().all()
interior = V.complement_dofs(boundary)


@LinearForm
def load(v, w):
    return v * apply(f, w.x)


b = asm(load, V)


@BilinearForm
def diffusion_form(u, v, w):
    return sum(v.grad * (q(w["w"]) * u.grad))


def diffusion_matrix(u):
    return asm(diffusion_form, V, w=V.interpolate(u))


dirichlet = apply(u_exact, mesh.p)  # P1 nodal interpolation
plot(V, dirichlet).get_figure().savefig(str(output_dir.joinpath("exact.png")))


def residual(u):
    r = b - diffusion_matrix(u) @ u
    r[boundary] = 0.0
    return r
```

```
u = np.zeros(V.N)
u[boundary] = dirichlet[boundary]
result = root(residual, u, method="krylov")

if result.success:
    u = result.x
    print("Success.  Residual =", np.linalg.norm(residual(u), np.inf))
    print("Nodal Linf error =", np.linalg.norm(u - dirichlet, np.inf))
    plot(V, u).get_figure().savefig(str(output_dir.joinpath("solution.png")))
else:
    print(result)
```

```
Success.  Residual = 1.4058151617812875e-07
Nodal Linf error = 1.2228777324096995e-08
```





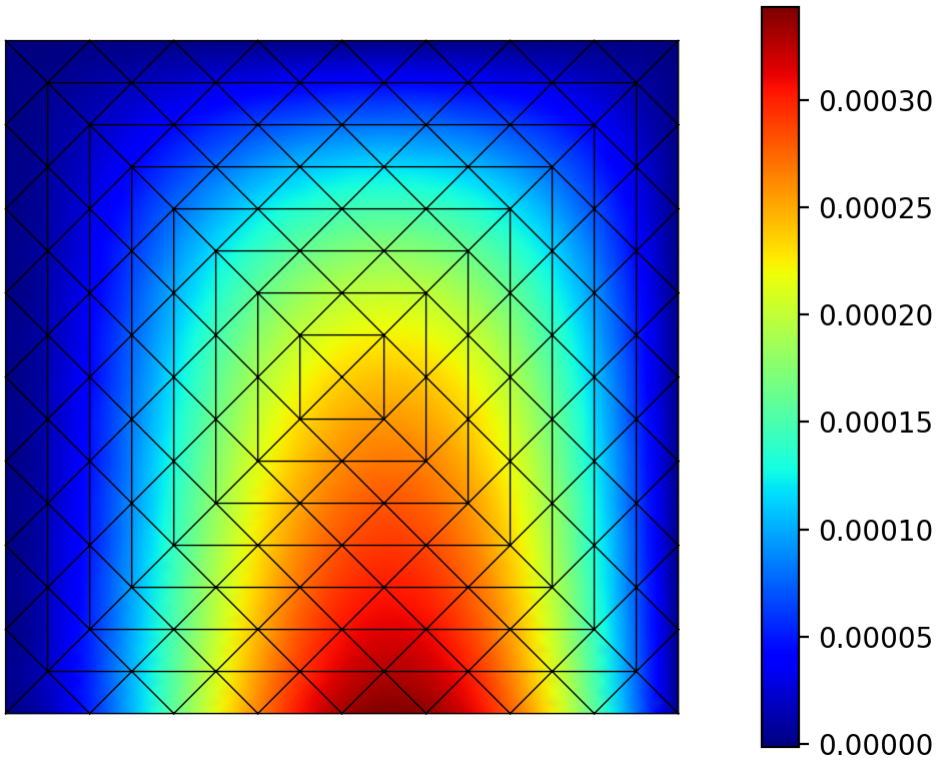
Fenics example #8 below. Note: In this case it is necessary to set the inline backend to "retina", as the .svg backend scrambles the graphic. (Whatever it is, "retina" matches vector quality.)

In [9]:

```
r"""
This example demonstrates the solution of a slightly more complicated problem
with multiple boundary conditions and a fourth-order differential operator. We
consider the `Kirchhoff plate bending problem
<https://en.wikipedia.org/wiki/Kirchhoff%E2%80%93Love_plate_theory>`_ which
finds its applications in solid mechanics. For a stationary plate of constant
thickness :math:`d`, the governing equation reads: find the deflection :math:`u
: \Omega \rightarrow \mathbb{R}` that satisfies
.. math::
    \frac{Ed^3}{12(1-\nu^2)} \Delta^2 u = f \quad \text{in $\Omega$},
where :math:`\Omega = (0,1)^2`, :math:`f` is a perpendicular force,
:math:`E` and :math:`\nu` are material parameters.
In this example, we analyse a :math:`1\,\text{m}^2` plate of steel with thickness :
The Young's modulus of steel is :math:`E = 200 \cdot 10^9\,\text{Pa}` and Poisson
ratio :math:`\nu = 0.3`.
In reality, the operator
.. math::
    \frac{Ed^3}{12(1-\nu^2)} \Delta^2
is a combination of multiple first-order operators:
.. math::
    \boldsymbol{K}(u) = - \boldsymbol{\varepsilon}(\nabla u), \quad \boldsymbol{\va
.. math::
    \boldsymbol{M}(u) = \frac{d^3}{12} \mathbb{C} \boldsymbol{K}(u), \quad \mathbb{
where :math:`\boldsymbol{I}` is the identity matrix. In particular,
```

```
.. math::
    \frac{Ed^3}{12(1-\nu^2)} \Delta^2 u = - \text{div}\,\textbf{div}\,\boldsymbol{M
There are several boundary conditions that the problem can take.
The *fully clamped* boundary condition reads
.. math::
    u = \frac{\partial u}{\partial \boldsymbol{n}} = 0,
where :math:`\boldsymbol{n}` is the outward normal.
Moreover, the *simply supported* boundary condition reads
.. math::
    u = 0, \quad M_{nn}(u)=0,
where :math:`M_{nn} = \boldsymbol{n} \cdot (\boldsymbol{M} \boldsymbol{n})`.
Finally, the *free* boundary condition reads
.. math::
    M_{nn}(u)=0, \quad V_{n}(u)=0,
where :math:`V_n` is the `Kirchhoff shear force <https://arxiv.org/pdf/1707.08396.p
definition is not needed here as this boundary condition is a
natural one.
The correct weak formulation for the problem is: find :math:`u \in V` such that
.. math::
    \int_\Omega \boldsymbol{M}(u) : \boldsymbol{K}(v) \,\mathrm{d}x = \int_\Omega f
where :math:`V` is now a subspace of :math:`H^2` with the essential boundary
conditions for :math:`u` and :math:`\frac{\partial u}{\partial \boldsymbol{n}}`.
Instead of constructing a subspace for :math:`H^2`, we discretise the problem
using the `non-conforming Morley finite element
<https://users.aalto.fi/~jakke74/WebFiles/Slides-Niiranen-ADMOS-09.pdf>`_ which
is a piecewise quadratic :math:`C^0`-continuous element for biharmonic problems.
The full source code of the example reads as follows:
.. literalinclude:: examples/ex02.py
    :start-after: EOF"""
from skfem import *
from skfem.models.poisson import unit_load
import numpy as np


m = (
    MeshTri.init_symmetric()
    .refined(3)
    .with_boundaries(
        {
            "left": lambda x: x[0] == 0,
            "right": lambda x: x[0] == 1,
            "top": lambda x: x[1] == 1,
        }
    )
)

e = ElementTriMorley()
ib = Basis(m, e)


@BilinearForm
def bilinf(u, v, w):
    from skfem.helpers import dd, ddot, trace, eye
    d = 0.1
    E = 200e9
    nu = 0.3
```

```
    def C(T):
        return E / (1 + nu) * (T + nu / (1 - nu) * eye(trace(T), 2))

    return d**3 / 12.0 * ddot(C(dd(u)), dd(v))


K = asm(bilinf, ib)
f = 1e6 * asm(unit_load, ib)

D = np.hstack([ib.get_dofs("left"), ib.get_dofs({"right", "top"}).all("u")])

x = solve(*condense(K, f, D=D))

def visualize():
    from skfem.visuals.matplotlib import draw, plot
    %config InlineBackend.figure_formats = ['retina']
    ax = draw(m)
    return plot(ib,
                x,
                ax=ax,
                shading='gouraud',
                colorbar=True,
                nrefs=2)

if __name__ == "__main__":
    visualize().show()
```



In [6]:
```
r"""Linear elasticity.
This example solves the linear elasticity problem using trilinear elements.  The
weak form of the linear elasticity problem is defined in
:func:`skfem.models.elasticity.linear_elasticity`.
```

```
"""

import numpy as np
from skfem import *
from skfem.models.elasticity import linear_elasticity, lame_parameters

m = MeshHex().refined(3)
e1 = ElementHex1()
e = ElementVector(e1)
ib = Basis(m, e, MappingIsoparametric(m, e1), 3)

K = asm(linear_elasticity(*lame_parameters(1e3, 0.3)), ib)

dofs = {
    'left' : ib.get_dofs(lambda x: x[0] == 0.0),
    'right': ib.get_dofs(lambda x: x[0] == 1.0),
}

u = ib.zeros()
u[dofs['right'].nodal['u^1']] = 0.3

I = ib.complement_dofs(dofs)

u = solve(*condense(K, x=u, I=I))

sf = 1.0
m = m.translated(sf * u[ib.nodal_dofs])

if __name__ == "__main__":
    from os.path import splitext
    from sys import argv

    m.save(splitext(argv[0])[0] + '.vtk')
```