

VIENNA UNIVERSITY OF TECHNOLOGY

COMPUTATIONAL MATHEMATICS - SEMINARY

INSTITUTE OF ANALYSIS AND SCIENTIFIC COMPUTING

PDE Constrained Shape Optimization

Authors:

Camilo TELLO FACHIN
12127084

Paul GENEST
12131124

Supervisor:

Dr. Kevin STURM

September 19, 2022



TECHNISCHE
UNIVERSITÄT
WIEN

Vienna University of Technology

Zusammenfassung

Zusammenfassung in Deutsch!

Abstract

Abstract in English!

Contents

1	Introduction	2
2	The Stokes Equations in NGSolve	3
3	Shape Optimization	6
3.1	Shape Derivative	7
3.1.1	Descent Direction	7
3.2	The Lagrangian, State Equation and Geometric Constraints	8
3.2.1	Derivative	8
3.3	Deformation	9
3.4	Iteration	10
4	Results and Conclusion	12
	References	13
	Appendices	14
A	Python Code Listing	14
B	XMI Code Listing	15
C	MATLAB Code Listing	16

To Do's

- What's still to you do make your supervisor/prof happy?

1 Introduction

i want to cite [3] and also [1] and also [7] and also [2], A figure example and a text where I refer to figure 1 below! Additionally I need other citations like [5] as well as [4] and [6] yes yes yes or no? David Kempf

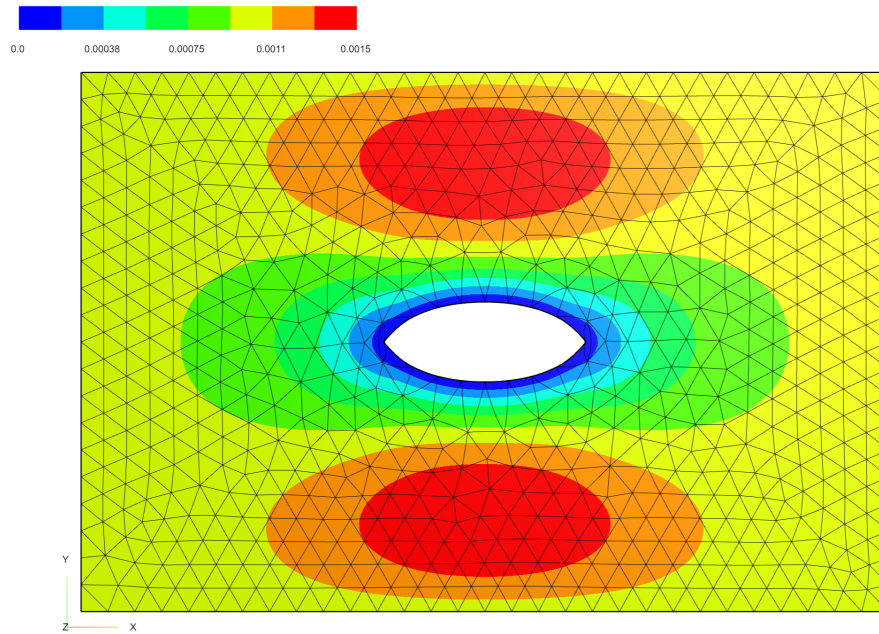
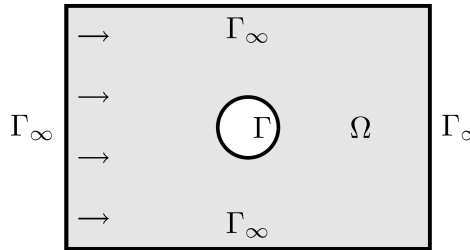


Figure 1: Velocity magnitude of Stokes flow after shape optimization

2 The Stokes Equations in NGSolve

The Stokes Equations are linear partial differential equations, which describe a stationary incompressible Newtonian fluid flow with high viscosities and low Reynolds numbers. For the implementation in NGSolve, a suitable geometry and boundary conditions are the ones proposed by Sturm et. al. [2], where the fluid flow around a cylinder is investigated while the outer boundary of Ω is prescribed a velocity strictly in x direction, the so called far field velocity:



$$\begin{aligned}
 -\mu \Delta u + \nabla p &= 0 & \text{in } \Omega, \\
 \operatorname{div} u &= 0 & \text{in } \Omega, \\
 \mathbf{u} &= 0 & \text{on } \Gamma, \\
 \mathbf{u} &= \mathbf{u}_\infty & \text{on } \Gamma_\infty,
 \end{aligned} \tag{1}$$

Figure 2: Domain Ω for Stokes PDE's (1) [2]

Where $\mu \in \mathbb{R}$ is the viscosity constant and is set to 1 for simplicity. The problem yields the vectorial velocity field $u : \Omega \rightarrow \mathbb{R}^d$ and the scalar pressure field $p : \Omega \rightarrow \mathbb{R}$. In order to solve the Stokes equation with the Finite Element Method in NGSolve, it needs to be transformed to the weak formulation, where the solutions u and p are linear combinations of basis functions in a Sobolev space. See Faustmann[5] Chapter 3 for further elaborations on Sobolev spaces. The weak formulation can be derived by multiplying the now called trial-functions u and p with test-functions v and q , perform transformations and integrate them. The test-functions have to fulfil certain conditions to permit the transformations in order to arrive at a weak problem with linear convergence rates, see Faustmann[5]:

Find $u \in [H_0^1(\Omega)]^d$ and $p \in L^2(\Omega)$ such that

$$\begin{aligned}
 \int_{\Omega} \nabla u : \nabla v \, dx + \int_{\Omega} \operatorname{div}(v) p \, dx &= 0 \quad \forall v \in [H_0^1(\Omega)]^d \\
 \int_{\Omega} \operatorname{div}(u) q \, dx &= 0 \quad \forall q \in L^2(\Omega)
 \end{aligned} \tag{2}$$

Instead of considering this as a system of equations, one can look at the mixed method as one variational problem on the product spaces $[H_0^1(\Omega)]^d \times L^2(\Omega)$, this is done by just adding both problems [5]:

$$\int_{\Omega} \nabla u : \nabla v \, dx + \int_{\Omega} \operatorname{div}(v) p \, dx + \int_{\Omega} \operatorname{div}(u) q \, dx = 0 \quad \forall (v, q) \in [H_0^1(\Omega)]^d \times L^2(\Omega) \tag{3}$$

In lines 19-26 of listing 2, the variational problem 3 is added to a `BilinearForm()`. After assembling of the system, in line 27-31 the non-zero Dirichlet conditions are assigned. When setting up the geometry, the boundaries already have to be named to do the boundary conditions assignment. The geometry shown in figure 2, is defined in the beginning in lines 5-11.

Basic Stokes PDE's with Python3 and NGSolve

```

1
2  from ngsolve import *
3  from netgen.geom2d import SplineGeometry
4  from ngsolve.webgui import Draw
5  # Geometry with meshwidth h_m
6  h_m = 0.4
7  geo = SplineGeometry()
8  geo.AddRectangle((-3,-2), (3, 2), bcs=("top", "out", "bot", "in"), leftdomain=1, rightdomain=0)
9  geo.AddCircle(c=(0, 0), r=0.5, leftdomain=0, rightdomain=1, bc="cyl", maxh=h_m)
10 mesh = Mesh(geo.GenerateMesh(maxh=h_m))
11 mesh.Curve(3);
12 # Setting up appropriate Function Spaces and boundary Conditions
13 k = 2
14 V = H1(mesh,order=k, dirichlet=("top|bot|cyl|in|out")
15 Q = H1(mesh,order=k-1)
16 FES = FESpace([V,V,Q]) # Omitting command VectorH1 --> [V,Q]
17 ux,uy,p = FES.TrialFunction()
18 vx,vy,q = FES.TestFunction()
19 # stokes equation
20 def Equation(ux,uy,p,vx,vy,q):
21     div_u = grad(ux)[0]+grad(uy)[1] # custom divergence u
22     div_v = grad(vx)[0]+grad(vy)[1] # custom divergence v
23     return (grad(ux)*grad(vx)+grad(uy)*grad(vy) + div_u*q + div_v*p)* dx
24 a = BilinearForm(FES)
25 a += Equation(ux,uy,p,vx,vy,q)
26 a.Assemble()
27 # Assign non-zero Dirichlet boundary conditions u_inf
28 gfu = GridFunction(FES)
29 uinf = 0.001
30 uinf.c = CoefficientFunction((uinf))
31 gfu.components[0].Set(uinf.c, definedon=mesh.Boundaries("in|top|bot|out"))
32 # Define Linear Equation System
33 def solveStokes():
34     res = gfu.vec.CreateVector()
35     res.data = -a.mat * gfu.vec
36     inv = a.mat.Inverse(FES.FreeDofs())
37     gfu.vec.data += inv * res
38     scene_state.Redraw()
39 # Solve LES and plot norm of u
40 solveStokes()
41 u_vec = CoefficientFunction((gfu.components[0], gfu.components[1]))
42 Draw(u_vec, mesh, "vel", draw_surf=True)

```


Below the solution obtained with NGSolve (see listing 2). On the surface of the cylinder, the no-slip condition (standard Dirichlet = 0) can be observed. Also an intuitive observation of the fulfilled continuity can be made: where the cross section is smaller, e.g. in x vicinity of the cylinder, the velocity is increased:

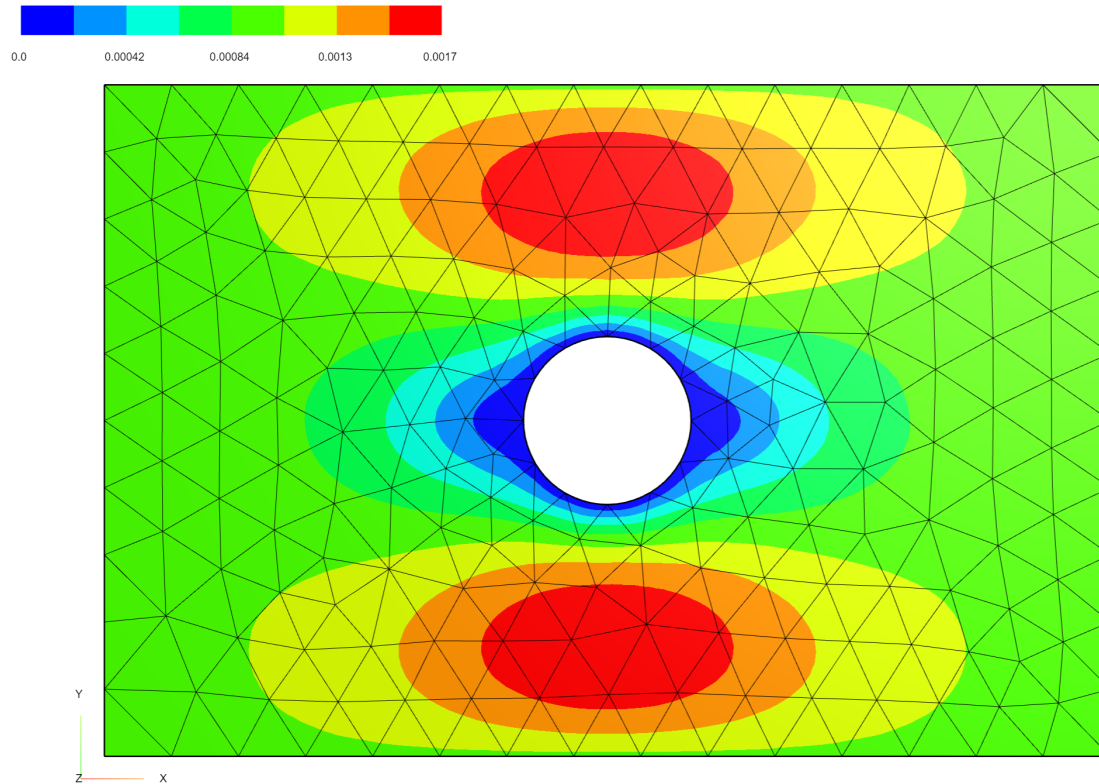


Figure 3: Surface Plot - Velocity $\|\mathbf{u}\|_2$ of Stokes Flow - FEM solution to problem (1)

3 Shape Optimization

Shape optimization is the process of minimizing of shape functions J . This function depends on the domain Ω , which will be perturbed in the minimization process. The perturbations of the shape Ω are described by the following transformation: $\Omega_t := (Id + tX)(\Omega)$. According to this, for small perturbations and $t > 0$, the shape derivative is: [3]

$$DJ(\Omega)(X) := \left(\frac{\partial}{\partial t} J(\Omega_t) \right) \Big|_{t=0} = \lim_{t \rightarrow 0} \frac{J(\Omega_t) - J(\Omega)}{t} \quad (4)$$

This is needed for ... ?

A lot of engineering applications require the shape to be dependent on a PDE. The resulting problems are called PDE constrained shape optimization. This yields a minimization problem of the shape function subjected to the side constraints of the (in our case) stokes equation.

This

3.1 Shape Derivative

The shape derivative is needed to find a shape that minimizes the given functional.

3.1.1 Descent Direction

To do some sort of gradient descent we need a negative descent direction. To achieve this we solve the deformation for a positive definite right hand side, which then can be subtracted to achieve a negative direction. The easiest solution for this is the use of the H^1 -norm.

3.2 The Lagrangian, State Equation and Geometric Constraints

It is shown in Sturm (2015a) that the shape derivative for a nonlinear PDEconstrained shape optimization problem can be computed as the derivative of the Lagrangian with respect to the perturbation parameter.[3]

This Lagrangian is given by the stokes equation with right hand side zero and the shape function J . For the Stokes equation there are only u and p used, not v and q .

$$\int_{\Omega} \nabla u : \nabla u \, dx + \int_{\Omega} \operatorname{div}(u)p \, dx + \int_{\Omega} \operatorname{div}(u)p \, dx + \int_{\Omega} Du : Du \, dx \quad (5)$$

Additional to these, there are geometrical constraints which should be enforced: On the one hand the volume has to be constant, because otherwise the obstacle just shrinks to nothing. This would obviously minimize the dissipated energy, but not tell anything useful for the solution.

Secondly, the barycenter of the obstacle should stay constant, because otherwise it could just try to move out of the mesh to minimize the shape function.

Since in our approach only the mesh for the flow outside of the obstacle is generated, we assign those geometric constraints on this mesh and not on the obstacle itself. In general that does not change anything about the problems with these constraints in itself. This is because if the barycenter and volume of the surrounding mesh stay constant, the same is true for the obstacle.

Additional there are scaling parameters added. These are set to a small initial value to get as soon as possible to a deformed shape and later on (when the shape does not change that much anymore) penalize the geometric constraints more, to achieve the real desired shape with the enforced constraints.

To calculate the volume is just integrating the 1 function over the entire domain. $vol = \int_{\Omega} 1 \, dx$ The barycenter has to be calculated in each possible direction of the domain, the formula for one is: $bc_{xi} = \frac{1}{vol} \int_{\Omega} x_i \, dx$

The formulas are used to calculate the initial value (with the subscript zero) and penalize any deviation from it with a number greater than zero, to steer away from shapes that do not comply these constraints. That is why the difference is squared.

$$\alpha(vol - vol_0)^2 + \beta(bc_x - bc_{x0})^2 + \beta(bc_y - bc_{y0})^2 \quad (6)$$

3.2.1 Derivative

After we set the Lagrangian up accordingly, we have to calculate its shape derivative. In our code for 5 this is done by using ngsolves implementation of `DiffShape()`. This way we only have to specify what is written in 5 and the rest is done by ngsolve on its own.

This does not work for the side constraints, because squaring and subtracting with these defined integrals is not straight forward. To calculate these by hand, one arrives at the following formulas for the volume:

$$2\alpha(vol - vol_0) * \operatorname{div}(X) \quad (7)$$

and the barycenter in direction x :

$$2\beta(bc_x - bc_{x0}) \cdot \left(-\frac{1}{vol^2} \int \operatorname{div}(X) \, dx \cdot \int x \, dx + \frac{1}{vol} \cdot \left(\int \operatorname{div}(X) * x \, dx + \int X_i \, dx \right) \right) \quad (8)$$

where x is the current direction variable and X_i is the deformation field in the i 'th (current) direction.

3.3 Deformation

The deformation, as described in the mathematical sense is the X in $\Omega_t := (Id + tX)(\Omega)$. Its according space is $H^1(\text{mesh}, \text{order}=2, \text{dim}=2)$, with dirichlet boundaries on the outsides of the square.

This variable in our ngsolve implementation is called **gfx**. This is a two-dimensional **GridFunction** which is computed from the shape derivative and stores the information how to deform the mesh at each node. The shape derivative is the linear form **dJ0mega** in our code. Since the goal is, to make steps in a descent direction (negative), we have to make sure we only calculate according solutions. This is done by setting the bilinear form of this problem to the H^1 norm.

$$\partial J(\Omega) = \int (\phi \cdot X) + (\nabla \phi \cdot \nabla X) dx \quad (9)$$

This always yields a positive value, if we subtract instead of add this value, we minimize the shape function.

Since the minimization is done in iterations, we have to keep track of the previous deformations. This is done in ngsolve by adding the parts of **gfx** to another variable called **gfset**. This **gfset** is then always used to call **SetDeformation()** on the mesh. With each call, this adds the **gfset** onto the mesh. To circumvent this, after each iteration **UnsetDeformation()** is called.

To not overshoot anything, instead of adding the entire **gfx** to **gfset**, it is scaled by a number divided by its norm. That way we can make sure, that each iteration deforms the mesh in small, similar sized steps. There is still one inconsistency: the **gfx** can also deform nodes inside the mesh, which change nothing for the real solution, but count towards this norm. That problem is solvable, by integrating this **GridFunction** over its boundary. Since the outside of our square are dirichlet boundaries, this way only the changes on the obstacle are measured. Another important thing to be aware of is, that symmetric deformations around an obstacle might cancel each other out in the integral. This is circumvented by calculating with the squared values.

3.4 Iteration

In this chapter we want to describe the process of one iteration which gets repeated time after time until the optimal shape is achieved.

There are a few parameters that are recalculated or conditionally updated through the iterations. This includes the current volume and barycenter values, their respective scalings α and β , and *gamma* for the Cauchy Riemann terms that ensure a better mesh quality.

Before we start with the iteration process we reset all possible variables: This includes the **gfset**, resetting the scene, and to reinitialize the parameters for the geometric constraints. If this step is skipped, one could start with weird variable values that lead to drastically different results.

Reset before iteration

```

1  gfset.Set((0,0))
2  mesh.SetDeformation(gfset)
3  scene.Redraw()
4
5  updateParams()
6  alpha0 = 1e-4
7  beta0 = 1e-0
8  gamma0 = 1e2
9  alpha.Set(alpha0)
10 beta.Set(beta0)
11 gamma.Set(gamma0)
12 iter_max = 750

```

This iteration is bounded by a maximum number of steps, even though it is also possible to take some measure on the **gfX** and determine a stop by it. This still requires parameter tuning, especially because some measures could yield different results depending on the mesh width. Other possibilities for breaks in the loop are a maximum number for the scaling parameters e.g. α or a very small difference in the drag from one iteration to the next.

Each iteration then starts by calling **SetDeformation(gfset)**, and redrawing needed scenes. This would also be the place to gather data, like the current drag or area of the mesh, etc.

Afterwards we start calculating towards the next step: We Assemble the state equation bilinear form and solve it over this newly deformed mesh. Following this we assemble the linear and bilinear form for the shape derivative and solve for a new deformation. After this is done, one can already use **UnsetDeformation()** and the last thing to do is updating values.

We scale the **gfX** to its desired magnitude and subtract it from **gfset**. At this point it is also wise to check for some measure of close we are to a good solution and for example increase the parameters for the geometric constraints.

The entire code inside the loop looks like this:

Iteration

```

1  mesh.SetDeformation(gfset)
2  scene.Redraw()
3  data.append(vol.Get())
4  a.Assemble()
5  solveStokes()
6  b.Assemble()
7  dJOmega.Assemble()
8  SolveDeformationEquation()

```

```
9      updateParams()
10     mesh.UnsetDeformation()
11     gfxnorm = Norm(gfX.vec)
12     scale = 0.01 / gfxnorm
13
14     if (gfxnorm < 1e-5):
15         if alpha.Get() < 1:
16             increaseParams(2, True)
17         else:
18             break
19     gfset.vec.data -= scale * gfX.vec
```

It is also possible to implement some sort of line search for the step size, similar to the armijo rule. This way one can do less iterations but take better/bigger steps, being more efficient in regards to computational efforts. More to this can be read in ...

4 Results and Conclusion

The implementation with automatic differentiation from `ngsolve` is not extremely complicated but not as straight forward as one would hope for. A big problem with this approach are the different types of objects in `ngsolve`, namely `float`, `CoefficientFunction`, `Parameter` and `SumOfIntegrals`, which are not easy to concatenate the right way to achieve the desirable function. This does not occur for the stokes equation or shape function, but complex side constraints are tougher than necessary.

The `DiffShape()` function helps to eliminate the need to derive and transform the terms in the Lagrangian ourselves, but is e.g. not usable for `VectorH1` functions. With the automatic Differentiation we were able to achieve the desired form: The volume constraint is very important for this piece and is possible to be implemented with and or without differentiation. This was also showed in a programming example, that this term on its own can deform any shape to the desired volume. On the other hand, the barycenter constraint does not hold up to its expectations. None of our implemented terms does give the desired functionality and others we are not able to implement in `ngsolve`.

Even without the barycenter constraint we were able to achieve this solution after 300 iterations:

The derived shape derivative of the paper [2] does not show the same behavior as the one created from `DiffShape()`. That means we were also not able to create a solution that looks similar to the one before.

References

- [1] V. Schulz and M. Siebenborn, “Computational comparison of surface metrics for PDE constrained Shape Optimization,” *Computational Methods in Applied Mathematics*, vol. 16, Sep. 2015. DOI: <https://doi.org/10.1515/cmam-2016-0009>.
- [2] J. Iglesias, K. Sturm, and F. Wechsung, “Two-Dimensional Shape Optimization with Nearly Conformal Transformations,” *SIAM Journal on Scientific Computing*, vol. 40, A3807–A3830, Jan. 2018. DOI: <https://doi.org/10.1137/17M1152711>.
- [3] P. Gangl, K. Sturm, M. Neunteufel, and J. Schöberl, “Fully and Semi-automated Shape Differentiation in NGSolve,” *Structural and multidisciplinary optimization*, vol. 63, no. 3, pp. 1579–1607, 2021. DOI: <https://doi.org/10.1007/s00158-020-02742-w>.
- [4] M. Faustmann, “Lecture notes for Applied Mathematics Foundations - TU Vienna ASC,” Feb. 2022.
- [5] M. Faustmann and J. Schoeberl, “Lecture notes for Numerical Methods for PDE’s - TU Vienna ASC,” Jun. 2022.
- [6] J. M. Melenk, “Lecture notes for Numerical Computation - TU Vienna ASC,” Feb. 2022.
- [7] K. Sturm, “Lecture notes for PDE constrained Optimization - TU Vienna ASC,” Jun. 2022.

A Python Code Listing

Here is an example of a python listing, you can change appearance of comments, strings, numbering, known commands and variables in the package settings in packages.tex. You can obviously use the listings environment in the rest of the document. The same procedure applies for listings in other languages.

Python Listing Title

```

1  ]
2  # Python Script, API Version = V18
3
4  import math
5
6  #  DELETE EVERYTHING -----
7
8  ClearAll ()
9
10 #  PARAMETERS -----
11
12 w = float(Parameters.w)      # side length of one element or half of a unit cell
13 e = float(Parameters.e)      # rectangle ratio e
14 b = w/(1+e)
15 rho = float(Parameters.rho)  # relative density
16 f = float(Parameters.f)      # number of layers = folds+1
17 h = 2*w/f                    # layer height = size of a unit cell divided by the number of layers
18 f = int(Parameters.f)
19
20 # Calculation of wall thickness t
21 t1 = ((math.sqrt(1-rho)+1)*math.sqrt(2)*w)/2
22 t2 = -((math.sqrt(1-rho)-1)*math.sqrt(2)*w)/2
23 if t1 < t2:
24     t=t1
25 else:
26     t=t2
27
28 # auxiliary variable to build up rectangle
29 m = math.sqrt(pow(t,2)*2)/2

```

B XMI Code Listing

Here is an example for XML code listing.

XML Listing Title

```
1 <extension version="1" name="EnergyIntegral" loadasdefault="True" >
2   <guid shortid="EnergyIntegral">8005c624-8869-4c74-b32b-97ac59c200b2</guid>
3   <script src="energy_integral.py" />
4   <interface context="Mechanical" >
```

C MATLAB Code Listing

Here is an example for MATLAB code listing

MATLAB Listing Title

```

1 %% Linear model Poly44 from MATLAB Curve Fit App:
2
3 %Polynomial Coefficients (with 95\% confidence bounds):
4     p00 =      13.79;  %(13.22, 14.36)
5     p10 =     -2.897; %(-3.454, -2.34)
6     p01 =      3.752; %(3.163, 4.34)
7     p20 =      3.279; %(2.231, 4.327)
8     p11 =      0.5404; %(-0.2001, 1.281)
9     p02 =      0.8638; %(-0.4624, 2.19)
10    p30 =      0.299; %(0.01281, 0.5851)
11    p21 =     -0.5091; %(-0.7299, -0.2884)
12    p12 =      0.4973; %(0.2716, 0.7229)
13    p03 =      0.3595; %(0.04484, 0.6741)
14    p40 =     -0.8495; %(-1.291, -0.4084)
15    p31 =     -0.02258; %(-0.3136, 0.2685)
16    p22 =     -0.2819; %(-0.5502, -0.01351)
17    p13 =      0.2674; %(-0.05265, 0.5874)
18    p04 =      0.2019; %(-0.3968, 0.8006)
19
20    f(x,y) = p00 + p10*x + p01*y + p20*x^2 + p11*x*y + p02*y^2 + p30*x^3 + p21*x^2*y
21    + p12*x*y^2 + p03*y^3 + p40*x^4 + p31*x^3*y + p22*x^2*y^2
22    + p13*x*y^3 + p04*y^4
23
24    %Goodness of fit:
25    %SSE: 3.189
26    %R-square: 0.9949
27    %Adjusted R-square: 0.9902
28    %RMSE: 0.4611

```