Behavior	Number of Parameters	Parameter Ordering	Restrictions
Represents a simple translation of points in the input space and has no effect on vectors or covariant vectors.	Same as the input space dimension.	The *i*-th parameter represents the translation in the *i*-th dimension.	Only defined when the input and output space have the same number of dimensions.

Table 3.3: Characteristics of the TranslationTransform class.

3.9.4 Translation Transform

The itk::TranslationTransform is probably the simplest yet one of the most useful transformations. It maps all Points by adding a Vector to them. Vector and covariant vectors remain unchanged under this transformation since they are not associated with a particular position in space. Translation is the best transform to use when starting a registration method. Before attempting to solve for rotations or scaling it is important to overlap the anatomical objects in both images as much as possible. This is done by resolving the translational misalignment between the images. Translations also have the advantage of being fast to compute and having parameters that are easy to interpret. The main characteristics of the translation transform are presented in Table 3.3.

3.9.5 Scale Transform

The itk::ScaleTransform represents a simple scaling of the vector space. Different scaling factors can be applied along each dimension. Points are transformed by multiplying each one of their coordinates by the corresponding scale factor for the dimension. Vectors are transformed in the same way as points. Covariant vectors, on the other hand, are transformed differently since anisotropic scaling does not preserve angles. Covariant vectors are transformed by *dividing* their components by the scale factor of the corresponding dimension. In this way, if a covariant vector was orthogonal to a vector, this orthogonality will be preserved after the transformation. The following equations summarize the effect of the transform on the basic geometric objects.

$$\begin{array}{lccccccc}
\text{Point} & \mathbf{P}' & = & T(\mathbf{P}) & : & \mathbf{P'}_\mathbf{i} & = & \mathbf{P_i} \cdot \mathbf{S_i} \\
\text{Vector} & \mathbf{V}' & = & T(\mathbf{V}) & : & \mathbf{V'}_\mathbf{i} & = & \mathbf{V_i} \cdot \mathbf{S_i} \\
\text{CovariantVector} & \mathbf{C}' & = & T(\mathbf{C}) & : & \mathbf{C'}_\mathbf{i} & = & \mathbf{C_i} / \mathbf{S_i}
\end{array} \tag{3.9}$$

where $\mathbf{P_i}$, $\mathbf{V_i}$ and $\mathbf{C_i}$ are the point, vector and covariant vector *i*-th components while $\mathbf{S_i}$ is the scaling

[7]Note that the term *Jacobian* is also commonly used for the matrix representing the derivatives of output point coordinates with respect to input point coordinates. Sometimes the term is loosely used to refer to the determinant of such a matrix. [17]

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Points are transformed by multiplying each one of their coordinates by the corresponding scale factor for the dimension. Vectors are transformed as points. Covariant vectors are transformed by *dividing* their components by the scale factor in the corresponding dimension.	Same as the input space dimension.	The i-th parameter represents the scaling in the i-th dimension.	Only defined when the input and output space have the same number of dimensions.

Table 3.4: Characteristics of the ScaleTransform class.

factor along dimension $i-th$. The following equation illustrates the effect of the scaling transform on a $3D$ point.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} S_1 & 0 & 0 \\ 0 & S_2 & 0 \\ 0 & 0 & S_3 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{3.10}$$

Scaling appears to be a simple transformation but there are actually a number of issues to keep in mind when using different scale factors along every dimension. There are subtle effects—for example, when computing image derivatives. Since derivatives are represented by covariant vectors, their values are not intuitively modified by scaling transforms.

One of the difficulties with managing scaling transforms in a registration process is that typical optimizers manage the parameter space as a vector space where addition is the basic operation. Scaling is better treated in the frame of a logarithmic space where additions result in regular multiplicative increments of the scale. Gradient descent optimizers have trouble updating step length, since the effect of an additive increment on a scale factor diminishes as the factor grows. In other words, a scale factor variation of $(1.0+\varepsilon)$ is quite different from a scale variation of $(5.0+\varepsilon)$.

Registrations involving scale transforms require careful monitoring of the optimizer parameters in order to keep it progressing at a stable pace. Note that some of the transforms discussed in following sections, for example, the AffineTransform, have hidden scaling parameters and are therefore subject to the same vulnerabilities of the ScaleTransform.

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Points are transformed by multiplying each one of their coordinates by the corresponding scale factor for the dimension. Vectors are transformed as points. Covariant vectors are transformed by *dividing* their components by the scale factor in the corresponding dimension.	Same as the input space dimension.	The *i*-th parameter represents the scaling in the *i*-th dimension.	Only defined when the input and output space have the same number of dimensions. The difference between this transform and the ScaleTransform is that here the scaling factors are passed as logarithms, in this way their behavior is closer to the one of a Vector space.

Table 3.5: Characteristics of the ScaleLogarithmicTransform class.

In cases involving misalignments with simultaneous translation, rotation and scaling components it may be desirable to solve for these components independently. The main characteristics of the scale transform are presented in Table 3.4.

3.9.6 Scale Logarithmic Transform

The `itk::ScaleLogarithmicTransform` is a simple variation of the `itk::ScaleTransform`. It is intended to improve the behavior of the scaling parameters when they are modified by optimizers. The difference between this transform and the ScaleTransform is that the parameter factors are passed here as logarithms. In this way, multiplicative variations in the scale become additive variations in the logarithm of the scaling factors.

3.9.7 Euler2DTransform

`itk::Euler2DTransform` implements a rigid transformation in $2D$. It is composed of a plane rotation and a two-dimensional translation. The rotation is applied first, followed by the translation. The following equation illustrates the effect of this transform on a $2D$ point,

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix} \tag{3.11}$$

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a $2D$ rotation and a $2D$ translation. Note that the translation component has no effect on the transformation of vectors and covariant vectors.	3	The first parameter is the angle in radians and the last two parameters are the translation in each dimension.	Only defined for two-dimensional input and output spaces.

Table 3.6: Characteristics of the Euler2DTransform class.

where θ is the rotation angle and (T_x, T_y) are the components of the translation.

A challenging aspect of this transformation is the fact that translations and rotations do not form a vector space and cannot be managed as linearly independent parameters. Typical optimizers make the loose assumption that parameters exist in a vector space and rely on the step length to be small enough for this assumption to hold approximately.

In addition to the non-linearity of the parameter space, the most common difficulty found when using this transform is the difference in units used for rotations and translations. Rotations are measured in radians; hence, their values are in the range $[-\pi, \pi]$. Translations are measured in millimeters and their actual values vary depending on the image modality being considered. In practice, translations have values on the order of 10 to 100. This scale difference between the rotation and translation parameters is undesirable for gradient descent optimizers because they deviate from the trajectories of descent and make optimization slower and more unstable. In order to compensate for these differences, ITK optimizers accept an array of scale values that are used to normalize the parameter space.

Registrations involving angles and translations should take advantage of the scale normalization functionality in order to obtain the best performance out of the optimizers. The main characteristics of the Euler2DTransform class are presented in Table 3.6.

3.9.8 CenteredRigid2DTransform

`itk::CenteredRigid2DTransform` implements a rigid transformation in $2D$. The main difference between this transform and the `itk::Euler2DTransform` is that here we can specify an arbitrary center of rotation, while the Euler2DTransform always uses the origin of the coordinate system as the center of rotation. This distinction is quite important in image registration since ITK images usually have their origin in the corner of the image rather than the middle. Rotational mis-registrations usually exist, however, as rotations around the center of the image, or at least as rotations around a

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a $2D$ rotation around a user-provided center followed by a $2D$ translation.	5	The first parameter is the angle in radians. Second and third are the center of rotation coordinates and the last two parameters are the translation in each dimension.	Only defined for two-dimensional input and output spaces.

Table 3.7: Characteristics of the CenteredRigid2DTransform class.

point in the middle of the anatomical structure captured by the image. Using gradient descent optimizers, it is almost impossible to solve non-origin rotations using a transform with origin rotations since the deep basin of the real solution is usually located across a high ridge in the topography of the cost function.

In practice, the user must supply the center of rotation in the input space, the angle of rotation and a translation to be applied after the rotation. With these parameters, the transform initializes a rotation matrix and a translation vector that together perform the equivalent of translating the center of rotation to the origin of coordinates, rotating by the specified angle, translating back to the center of rotation and finally translating by the user-specified vector.

As with the Euler2DTransform, this transform suffers from the difference in units used for rotations and translations. Rotations are measured in radians; hence, their values are in the range $[-\pi,\pi]$. The center of rotation and the translations are measured in millimeters, and their actual values vary depending on the image modality being considered. Registrations involving angles and translations should take advantage of the scale normalization functionality of the optimizers in order to get the best performance out of them.

The following equation illustrates the effect of the transform on an input point (x,y) that maps to the output point (x',y'),

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x - C_x \\ y - C_y \end{bmatrix} + \begin{bmatrix} T_x + C_x \\ T_y + C_y \end{bmatrix} \tag{3.12}$$

where θ is the rotation angle, (C_x,C_y) are the coordinates of the rotation center and (T_x,T_y) are the components of the translation. Note that the center coordinates are subtracted before the rotation and added back after the rotation. The main features of the CenteredRigid2DTransform are presented in Table 3.7.

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a $2D$ rotation, homogeneous scaling and a $2D$ translation. Note that the translation component has no effect on the transformation of vectors and covariant vectors.	4	The first parameter is the scaling factor for all dimensions, the second is the angle in radians, and the last two parameters are the translations in (x,y) respectively.	Only defined for two-dimensional input and output spaces.

Table 3.8: Characteristics of the Similarity2DTransform class.

3.9.9 Similarity2DTransform

The itk::Similarity2DTransform can be seen as a rigid transform combined with an isotropic scaling factor. This transform preserves angles between lines. In its $2D$ implementation, the four parameters of this transformation combine the characteristics of the itk::ScaleTransform and itk::Euler2DTransform. In particular, those relating to the non-linearity of the parameter space and the non-uniformity of the measurement units. Gradient descent optimizers should be used with caution on such parameter spaces since the notions of gradient direction and step length are ill-defined.

The following equation illustrates the effect of the transform on an input point (x,y) that maps to the output point (x',y'),

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x - C_x \\ y - C_y \end{bmatrix} + \begin{bmatrix} T_x + C_x \\ T_y + C_y \end{bmatrix} \tag{3.13}$$

where λ is the scale factor, θ is the rotation angle, (C_x,C_y) are the coordinates of the rotation center and (T_x,T_y) are the components of the translation. Note that the center coordinates are subtracted before the rotation and scaling, and they are added back afterwards. The main features of the Similarity2DTransform are presented in Table 3.8.

A possible approach for controlling optimization in the parameter space of this transform is to dynamically modify the array of scales passed to the optimizer. The effect produced by the parameter scaling can be used to steer the walk in the parameter space (by giving preference to some of the parameters over others). For example, perform some iterations updating only the rotation angle, then balance the array of scale factors in the optimizer and perform another set of iterations updating only the translations.

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a $3D$ rotation and a $3D$ translation. The rotation is specified as a quaternion, defined by a set of four numbers $\mathbf{q}$. The relationship between quaternion and rotation about vector $\mathbf{n}$ by angle θ is as follows: $\mathbf{q} = (\mathbf{n}\sin(\theta/\mathbf{2}), \cos(\theta/\mathbf{2}))$ Note that if the quaternion is not of unit length, scaling will also result.	7	The first four parameters defines the quaternion and the last three parameters the translation in each dimension.	Only defined for three-dimensional input and output spaces.

Table 3.9: Characteristics of the QuaternionRigidTransform class.

3.9.10 QuaternionRigidTransform

The `itk::QuaternionRigidTransform` class implements a rigid transformation in $3D$ space. The rotational part of the transform is represented using a quaternion while the translation is represented with a vector. Quaternions components do not form a vector space and hence raise the same concerns as the `itk::Similarity2DTransform` when used with gradient descent optimizers.

The `itk::QuaternionRigidTransformGradientDescentOptimizer` was introduced into the toolkit to address these concerns. This specialized optimizer implements a variation of a gradient descent algorithm adapted for a quaternion space. This class ensures that after advancing in any direction on the parameter space, the resulting set of transform parameters is mapped back into the permissible set of parameters. In practice, this comes down to normalizing the newly-computed quaternion to make sure that the transformation remains rigid and no scaling is applied. The main characteristics of the QuaternionRigidTransform are presented in Table 3.9.

The Quaternion rigid transform also accepts a user-defined center of rotation. In this way, the transform can easily be used for registering images where the rotation is mostly relative to the center of the image instead of one of the corners. The coordinates of this rotation center are not subject to optimization. They only participate in the computation of the mappings for Points and in the computation of the Jacobian. The transformations for Vectors and CovariantVector are not affected by the selection of the rotation center.

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a $3D$ rotation. The rotation is specified by a versor or unit quaternion. The rotation is performed around a user-specified center of rotation.	3	The three parameters define the versor.	Only defined for three-dimensional input and output spaces.

Table 3.10: Characteristics of the Versor Transform

3.9.11 VersorTransform

By definition, a *Versor* is the rotational part of a Quaternion. It can also be defined as a *unit-quaternion* [24, 27]. Versors only have three independent components, since they are restricted to reside in the space of unit-quaternions. The implementation of versors in the toolkit uses a set of three numbers. These three numbers correspond to the first three components of a quaternion. The fourth component of the quaternion is computed internally such that the quaternion is of unit length. The main characteristics of the `itk::VersorTransform` are presented in Table 3.10.

This transform exclusively represents rotations in $3D$. It is intended to rapidly solve the rotational component of a more general misalignment. The efficiency of this transform comes from using a parameter space of reduced dimensionality. Versors are the best possible representation for rotations in $3D$ space. Sequences of versors allow the creation of smooth rotational trajectories; for this reason, they behave stably under optimization methods.

The space formed by versor parameters is not a vector space. Standard gradient descent algorithms are not appropriate for exploring this parameter space. An optimizer specialized for the versor space is available in the toolkit under the name of `itk::VersorTransformOptimizer`. This optimizer implements versor derivatives as originally defined by Hamilton [24].

The center of rotation can be specified by the user with the `SetCenter()` method. The center is not part of the parameters to be optimized, therefore it remains the same during an optimization process. Its value is used during the computations for transforming Points and when computing the Jacobian.

3.9.12 VersorRigid3DTransform

The `itk::VersorRigid3DTransform` implements a rigid transformation in $3D$ space. It is a variant of the `itk::QuaternionRigidTransform` and the `itk::VersorTransform`. It can be seen as a `itk::VersorTransform` plus a translation defined by a vector. The advantage of this class with

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a $3D$ rotation and a $3D$ translation. The rotation is specified by a versor or unit quaternion, while the translation is represented by a vector. Users can specify the coordinates of the center of rotation.	6	The first three parameters define the versor and the last three parameters the translation in each dimension.	Only defined for three-dimensional input and output spaces.

Table 3.11: Characteristics of the VersorRigid3DTransform class.

respect to the QuaternionRigidTransform is that it exposes only six parameters, three for the versor components and three for the translational components. This reduces the search space for the optimizer to six dimensions instead of the seven dimensional used by the QuaternionRigidTransform. This transform also allows the users to set a specific center of rotation. The center coordinates are not modified during the optimization performed in a registration process. The main features of this transform are summarized in Table 3.11. This transform is probably the best option to use when dealing with rigid transformations in $3D$.

Given that the space of Versors is not a Vector space, typical gradient descent optimizers are not well suited for exploring the parametric space of this transform. The `itk::VersorRigid3DTranformOptimizer` has been introduced in the ITK toolkit with the purpose of providing an optimizer that is aware of the Versor space properties on the rotational part of this transform, as well as the Vector space properties on the translational part of the transform.

3.9.13 Euler3DTransform

The `itk::Euler3DTransform` implements a rigid transformation in $3D$ space. It can be seen as a rotation followed by a translation. This class exposes six parameters, three for the Euler angles that represent the rotation and three for the translational components. This transform also allows the users to set a specific center of rotation. The center coordinates are not modified during the optimization performed in a registration process. The main features of this transform are summarized in Table 3.12.

Three rotational parameters are non-linear and do not behave like Vector spaces. This must be taken into account when selecting an optimizer to work with this transform and when fine tuning the parameters of the optimizer. It is strongly recommended to use this transform by introducing very

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a rigid rotation in $3D$ space. That is, a rotation followed by a $3D$ translation. The rotation is specified by three angles representing rotations to be applied around the X, Y and Z axes one after another. The translation part is represented by a Vector. Users can also specify the coordinates of the center of rotation.	6	The first three parameters are the rotation angles around X, Y and Z axes, and the last three parameters are the translations along each dimension.	Only defined for three-dimensional input and output spaces.

Table 3.12: Characteristics of the Euler3DTransform class.

small variations on the rotational components. A small rotation will be in the range of 1 degree, which in radians is approximately 0.01745.

You should not expect this transform to be able to compensate for large rotations just by being driven with the optimizer. In practice you must provide a reasonable initialization of the transform angles and only need to correct for residual rotations in the order of 10 or 20 degrees.

3.9.14 Similarity3DTransform

The `itk::Similarity3DTransform` implements a similarity transformation in $3D$ space. It can be seen as an homogeneous scaling followed by a `itk::VersorRigid3DTransform`. This class exposes seven parameters: one for the scaling factor, three for the versor components and three for the translational components. This transform also allows the user to set a specific center of rotation. The center coordinates are not modified during the optimization performed in a registration process. Both the rotation and scaling operations are performed with respect to the center of rotation. The main features of this transform are summarized in Table 3.13.

The scaling and rotational spaces are non-linear and do not behave like Vector spaces. This must be taken into account when selecting an optimizer to work with this transform and when fine tuning the parameters of the optimizer.

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a $3D$ rotation, a $3D$ translation and homogeneous scaling. The scaling factor is specified by a scalar, the rotation is specified by a versor, and the translation is represented by a vector. Users can also specify the coordinates of the center of rotation, which is the same center used for scaling.	7	The first three parameters define the Versor, the next three parameters the translation in each dimension, and the last parameter is the isotropic scaling factor.	Only defined for three-dimensional input and output spaces.

Table 3.13: Characteristics of the Similarity3DTransform class.

3.9.15 Rigid3DPerspectiveTransform

The `itk::Rigid3DPerspectiveTransform` implements a rigid transformation in $3D$ space followed by a perspective projection. This transform is intended to be used in $3D/2D$ registration problems where a 3D object is projected onto a 2D plane. This is the case in Fluoroscopic images used for image-guided intervention, and it is also the case for classical radiography. Users must provide a value for the focal distance to be used during the computation of the perspective transform. This transform also allows users to set a specific center of rotation. The center coordinates are not modified during the optimization performed in a registration process. The main features of this transform are summarized in Table 3.14. This transform is also used when creating Digitally Reconstructed Radiographs (DRRs).

The strategies for optimizing the parameters of this transform are the same ones used for optimizing the VersorRigid3DTransform. In particular, you can use the same VersorRigid3DTranform-Optimizer in order to optimize the parameters of this class.

3.9.16 AffineTransform

The `itk::AffineTransform` is one of the most popular transformations used for image registration. Its main advantage comes from its representation as a linear transformation. The main features

Behavior	Number of Parameters	Parameter Ordering	Restrictions
Represents a rigid $3D$ transformation followed by a perspective projection. The rotation is specified by a Versor, while the translation is represented by a Vector. Users can specify the coordinates of the center of rotation. They must specify a focal distance to be used for the perspective projection. The rotation center and the focal distance parameters are not modified during the optimization process.	6	The first three parameters define the Versor and the last three parameters the Translation in each dimension.	Only defined for three-dimensional input and two-dimensional output spaces. This is one of the few transforms where the input space has a different dimension from the output space.

Table 3.14: Characteristics of the Rigid3DPerspectiveTransform class.

Behavior	Number of Parameters	Parameter Ordering	Restrictions
Represents an affine transform composed of rotation, scaling, shearing and translation. The transform is specified by a $N \times N$ matrix and a $N \times 1$ vector where N is the space dimension.	$(N+1) \times N$	The first $N \times N$ parameters define the matrix in column-major order (where the column index varies the fastest). The last N parameters define the translations for each dimension.	Only defined when the input and output space have the same dimension.

Table 3.15: Characteristics of the AffineTransform class.

of this transform are presented in Table 3.15.

The set of AffineTransform coefficients can actually be represented in a vector space of dimension $(N+1)\times N$. This makes it possible for optimizers to be used appropriately on this search space. However, the high dimensionality of the search space also implies a high computational complexity of cost-function derivatives. The best compromise in the reduction of this computational time is to use the transform's Jacobian in combination with the image gradient for computing the cost-function derivatives.

The coefficients of the $N\times N$ matrix can represent rotations, anisotropic scaling and shearing. These coefficients are usually of a very different dynamic range compared to the translation coefficients. Coefficients in the matrix tend to be in the range $[-1:1]$, but are not restricted to this interval. Translation coefficients, on the other hand, can be on the order of 10 to 100, and are basically related to the image size and pixel spacing.

This difference in scale makes it necessary to take advantage of the functionality offered by the optimizers for rescaling the parameter space. This is particularly relevant for optimizers based on gradient descent approaches. This transform lets the user set an arbitrary center of rotation. The coordinates of the rotation center do not make part of the parameters array passed to the optimizer. Equation 3.14 illustrates the effect of applying the AffineTransform to a point in $3D$ space.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} M_{00} & M_{01} & M_{02} \\ M_{10} & M_{11} & M_{12} \\ M_{20} & M_{21} & M_{22} \end{bmatrix} \cdot \begin{bmatrix} x-C_x \\ y-C_y \\ z-C_z \end{bmatrix} + \begin{bmatrix} T_x+C_x \\ T_y+C_y \\ T_z+C_z \end{bmatrix} \tag{3.14}$$

A registration based on the affine transform may be more effective when applied after simpler transformations have been used to remove the major components of misalignment. Otherwise it will incur an overwhelming computational cost. For example, using an affine transform, the first set of optimization iterations would typically focus on removing large translations. This task could instead be accomplished by a translation transform in a parameter space of size N instead of the $(N+1)\times N$ associated with the affine transform.

Tracking the evolution of a registration process that uses AffineTransforms can be challenging, since it is difficult to represent the coefficients in a meaningful way. A simple printout of the transform coefficients generally does not offer a clear picture of the current behavior and trend of the optimization. A better implementation uses the affine transform to deform a wire-frame cube which is shown in a $3D$ visualization display.

3.9.17 BSplineDeformableTransform

The `itk::BSplineDeformableTransform` is designed to be used for solving deformable registration problems. This transform is equivalent to generating a deformation field where a deformation vector is assigned to every point in space. The deformation vectors are computed using BSpline interpolation from the deformation values of points located in a coarse grid, which is usually referred to as the BSpline grid.

Behavior	**Number of Parameters**	**Parameter Ordering**	**Restrictions**
Represents a free-form deformation by providing a deformation field from the interpolation of deformations in a coarse grid.	$M \times N$	Where M is the number of nodes in the BSpline grid and N is the dimension of the space.	Only defined when the input and output space have the same dimension. This transform has the advantage of being able to compute deformable registration. It also has the disadvantage of a very high-dimensional parametric space, and therefore requiring long computation times.

Table 3.16: Characteristics of the BSplineDeformableTransform class.

The BSplineDeformableTransform is not flexible enough to account for large rotations or shearing, or scaling differences. In order to compensate for this limitation, it provides the functionality of being composed with an arbitrary transform. This transform is known as the *Bulk* transform and it applied to points before they are mapped with the displacement field.

This transform does not provide functionality for mapping Vectors nor CovariantVectors—only Points can be mapped. This is because the variations of a vector under a deformable transform actually depend on the location of the vector in space. In other words, Vectors only make sense as the relative position between two points.

The BSplineDeformableTransform has a very large number of parameters and therefore is well suited for the `itk::LBFGSOptimizer` and `itk::LBFGSBOptimizer`. The use of this transform was proposed in the following papers [52, 39, 40].

3.9.18 KernelTransforms

Kernel Transforms are a set of Transforms that are also suitable for performing deformable registration. These transforms compute on-the-fly the displacements corresponding to a deformation field. The displacement values corresponding to every point in space are computed by interpolation from the vectors defined by a set of *Source Landmarks* and a set of *Target Landmarks*.

Several variations of these transforms are available in the toolkit. They differ in the type of interpolation kernel that is used when computing the deformation in a particular point of space. Note that these transforms are computationally expensive and that their numerical complexity is proportional to the number of landmarks and the space dimension.

The following is the list of Transforms based on the KernelTransform.

- `itk::ElasticBodySplineKernelTransform`
- `itk::ElasticBodyReciprocalSplineKernelTransform`
- `itk::ThinPlateSplineKernelTransform`
- `itk::ThinPlateR2LogRSplineKernelTransform`
- `itk::VolumeSplineKernelTransform`

Details about the mathematical background of these transform can be found in the paper by Davis *et. al* [14] and the papers by Rohr *et. al* [50, 51].

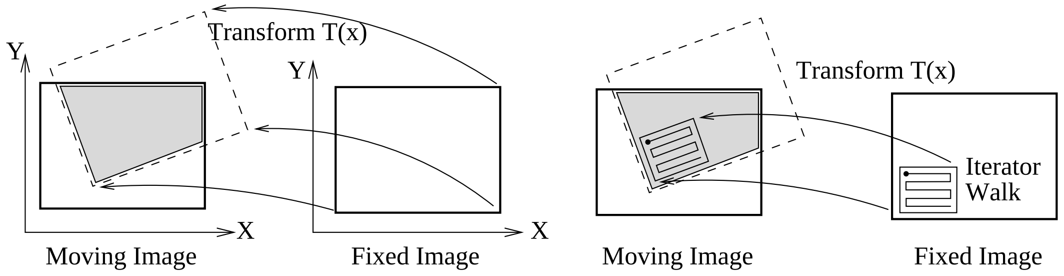

Figure 3.40: The moving image is mapped into the fixed image space under some spatial transformation. An iterator walks through the fixed image and its coordinates are mapped onto the moving image.

3.10 Interpolators

In the registration process, the metric typically compares intensity values in the fixed image against the corresponding values in the transformed moving image. When a point is mapped from one space to another by a transform, it will in general be mapped to a non-grid position. Therefore, interpolation is required to evaluate the image intensity at the mapped position.

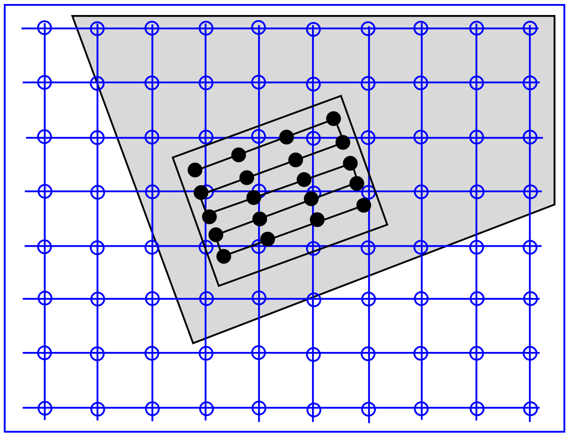

Figure 3.41: Grid positions of the fixed image map to non-grid positions of the moving image.

Figure 3.40 (left) illustrates the mapping of the fixed image space onto the moving image space. The transform maps points from the fixed image coordinate system onto the moving image coordinate system. The figure highlights the region of overlap between the two images after the mapping. The right side illustrates how an iterator is used to walk through a region of the fixed image. Each one of the iterator positions is mapped by the transform onto the moving image space in order to find the homologous pixel.

Figure 3.41 presents a detailed view of the mapping from the fixed image to the moving image. In general, the grid positions of the fixed image will not be mapped onto grid positions of the moving image. Interpolation is needed for estimating the intensity of the moving image at these non-grid positions. The service is provided in ITK by interpolator classes that can be plugged into the registration method.

The following interpolators are available: